VECPAR 98

3rd internacional meeting on
vector and parallel processing

1998
June, 21 - 23

Faculdade de Engenharia
da Universidade do Porto

Proceedings
Part I *(June 21)*

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	10 August 1998	Conference Proceedings

4. TITLE AND SUBTITLE

VECPAR 98 3rd International Meeting on Vector and Parallel Processing

5. FUNDING NUMBERS

F6170898W0009

6. AUTHOR(S)

Conference Committee

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Faculdade de Engenharia da Universidade do Porto
Seccao dos Bragas
Porto Codex 4099
Portugal

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

EOARD
PSC 802 BOX 14
FPO 09499-0200

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

CSP 98-1006

11. SUPPLEMENTARY NOTES

Consists of three volumes.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

A

13. ABSTRACT (Maximum 200 words)

The Final Proceedings for VECPAR 98 3rd International Meeting on Vector and Parallel Processing, 21 June 1998 - 23 June 1998

This is an interdisciplinary conference. Topics include parallel and distributed computing, image processing and synthesis, real-time and embedded systems.

14. SUBJECT TERMS

Computers, Signal Processing, Mathematics, Modelling & Simulation

15. NUMBER OF PAGES

1088

16. PRICE CODE

N/A

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

VECPAR'98

3rd International Meeting on Vector and Parallel Processing

1998, June 21-23

Conference Proceedings
Part I
(Sunday, June 21)

FEUP
Faculdade de Engenharia
da Universidade do Porto

Preface

This volume consists of the invited talks, papers and posters presented during the VECPAR'98 - 3rd International Meeting on Vector and Parallel Processing. The meeting, organised by the FEUP - Faculdade de Engenharia da Universidade do Porto (Faculty of Engineering of the University of Porto), is held at Fundação Dr. António Cupertino de Miranda, in Porto (Portugal), from 21 to 23 June, 1998.

VECPAR'98 is the third in a series of VECPAR meetings initiated in 1993 (VECPAR'93, VECPAR'96) on vector and parallel computing. The format of previous meetings was preserved and it was organised around scientific sessions initiated by thematic key invited lectures, followed by contributed papers. The 66 papers and 20 posters presented at the conference were the result of a selection from more than 120 extended abstracts originated from 27 countries.

It is our great pleasure to express our gratitude to all people that helped us during the preparation of this event, and in particular to the members of the Scientific Committee. Without their collaboration and prompt reviews it would have been impossible to fulfill the deadlines imposed by the organisation. Also, with the contribution and comments of the Scientific Committee, the authors had the opportunity to improve the original versions of their papers.

We are very grateful to all sponsors for their support, without which the VECPAR'98 would not have been possible.

Porto, June 1998 *The Organising and Scientific Committees Chairs*

Committees

Organizing Committee

- Lígia M. Ribeiro
- Augusto de Sousa

Advisory Committee

- F. Nunes Ferreira
- J. Carlos Lopes
- J. Silva Matos
- J. César Sá
- J. Marques dos Santos
- R. Moreira Vidal

Scientific Committee

- Chair: J. Palma (Portugal)
- Vice-Chairs: J. Dongarra (USA), V. Hernandez (Spain)

- P. Amestoy (France)
- E. Aurell (Sweden)
- A. Chalmers (England)
- A. Coutinho (Brazil)
- J. F. Cunha (Portugal)
- J. C. Cunha (Portugal)
- M. Daydé (France)
- J. Dekeyser (France)
- R. Delgado (Portugal)
- F. d'Almeida (Portugal)
- J. Duarte (Portugal)
- I. Duff (France, England)
- D. Falcão (Brazil)
- S. Gama (Portugal)
- L. Giraud (France)
- S. Hammarling (England)
- D. Heermann (Germany)
- W. Janke (Germany)
- D. Knight (USA)
- V. Kumar (USA)
- R. Lins (Brazil)
- J. C. Long (Spain)
- J. P. Lopes (Portugal)
- E. Luque (Spain)
- P. Marquet (France)
- P. de Miguel (Spain)
- F. Moura (Portugal)
- K. Nagel (USA)
- M. Novotny (USA)
- E. Oñate (Spain)
- A. Padilha (Portugal)
- R. Pandey (USA)
- M. Peric (Germany)
- J. Pereira (Portugal)
- H. Pina (Portugal)
- A. Proença (Portugal)
- R. Ralha (Portugal)
- Y. Robert (France)
- A. Ruano (Portugal)
- D. Ruiz (France)
- H. Ruskin (Ireland)
- F. Silva (Portugal)
- J. G. Silva (Portugal)
- F. Tirado (Spain)
- B. Tourancheau (France)
- V. Venkatakrishnan (USA)
- P. Veríssimo (Portugal)
- J. S. Wang (Singapore)
- E. Zapata (Spain)

Sponsoring Organizations

- Faculdade de Engenharia da Universidade do Porto

- Câmara Municipal do Porto

- *European Office of Aerospace Research and Development* (EOARD)

- Fundação Dr. António Cupertino de Miranda
- Fundação Luso-Americana para o Desenvolvimento
- Fundação para a Ciência e a Tecnologia

- Fundação para a Computação Científica Nacional

- Associação de Estudantes da Faculdade de Engenharia do Porto

- Bull

- Digital Equipment Portugal

- ICL

- Silicon Graphics

Table of Contents

Some Unusual Matrix Eigenvalue Problems

Zhaojun Bai[1] and Gene H. Golub[2]

[1] University of Kentucky, Lexington, KY 40506, USA,
bai@ms.uky.edu
[2] Stanford University, Stanford, CA 94305, USA
golub@sccm.stanford.edu

Abstract. We survey some unusual eigenvalue problems arising in different applications. We show that all these problems can be cast as problems of estimating quadratic forms. Numerical algorithms based on the well-known Gauss-type quadrature rules and Lanczos process are reviewed for computing these quadratic forms. These algorithms reference the matrix in question only through a matrix-vector product operation. Hence it is well suited for large sparse problems. Some selected numerical examples are presented to illustrate the efficiency of such an approach.

1 Introduction

Matrix eigenvalue problems play a significant role in many areas of computational science and engineering. It often happens that many eigenvalue problems arising in applications may not appear in a standard form that we usually learn from a textbook and find in software packages for solving eigenvalue problems. In this paper, we described some unusual eigenvalue problems we have encountered. Some of those problems have been studied in literature and some are new. We are particularly interested in solving those associated with large sparse problems. Many existing techniques are only suitable for dense matrix computations and becomes inadequate for large sparse problems.

We will show that all these unusal eigenvalue problems can be converted to the problem of computing a quadratic form $u^T f(A)u$, for a properly defined matrix A, a vector u and a function f. Numerical techniques for computing the quadratic form to be discussed in this paper will based on the work initially proposed in [6] and further developed in [11, 12, 2]. In this technique, we first transfer the problem of computing the quadratic form to a Riemann-Stieltjes integral problem, and then use Gauss-type quadrature rules to approximate the integral, which then brings the orthogonal polynomial theory and the underlying Lanczos procedure into the scene. This approach is well suitable for large sparse problems, since it references the matrix A through a user provided subroutine to form the matrix-vector product Ax.

The basic time-consuming kernels for computing quadratic forms using parallelism are vector inner products, vector updates and matrix-vector products: this is similar to most iterative methods in linear algebra. Vector inner products and updates can be easily parallelized: each processor computes the vector-vector operations of corresponding segments of vectors (local vector operations

(LVOs)), and if necessary, the results of LVOs have to sent to other processors to be combined for the global vector-vector operations. For the matrix-vector product, the user can either explore the particular structure of the matrix in question for parallelism, or split the matrix into strips corresponding to the vector segments. Each process then computes the matrix-vector product of one strip. Furthermore, the iterative loop of algorithms can be designed to overlap communication and computation and eliminating some of the synchronization points. The reader may see [8, 4] and references therein for further details.

The rest of the paper is organized as follows. Section 2 describes some unusual eigenvalue problems and shows that these problems can be converted to the problem of computing a quadratic form. Section 3 reviews numerical methods for computing a quadratic form. Section 4 shows that how these numerical methods can be applied to those problems described in section 2. Some selected numerical examples are presented in section 5. Concluding remarks are in section 5.

2 Some Unusual Matrix Eigenvalue Problems

2.1 Constrained eigenvalue problem

Let A be a real symmetric matrix of order N, and c a given N vector with $c^T c = 1$. We are interested in the following optimization problem

$$\max_x \quad x^T A x \tag{1}$$

subject to the constraints

$$x^T x = 1 \tag{2}$$

and

$$c^T x = 0. \tag{3}$$

Let

$$\phi(x, \lambda, \mu) = x^T A x - \lambda(x^T x - 1) + 2\mu x^T c, \tag{4}$$

where λ, μ are Lagrange multipliers. Differentiating (4) with respect to x, we are led to the equation

$$Ax - \lambda x + \mu c = 0.$$

Then

$$x = -\mu(A - \lambda I)^{-1} c.$$

Using the constraint (3), we have

$$c^T (A - \lambda I)^{-1} c = 0. \tag{5}$$

An equation of such type is referred as a *secular equation*. Now the problem becomes finding the largest λ of the above secular equation.

We note that in [10], the problem is cast as computing the largest eigenvalue of the matrix PAP, where P is a project matrix $P = I - cc^T$.

2.2 Modified eigenvalue problem

Let us consider solving the following eigenvalue problems

$$Ax = \lambda x$$

and

$$(A + cc^T)\bar{x} = \bar{\lambda}\bar{x}$$

where A is a symmetric matrix and c is a vector and without loss of generality, we assume $c^T c = 1$. The second eigenvalue problem can be regarded as a modifed or perturbed eigenvalue problem of the first one. We are interested in obtaining some, not all, of the eigenvalues of both problems. Such computation task often appears in structural dynamic (re-)analysis and other applications [5].

By simple algebraic derivation, it is easy to show that the eigenvalues $\bar{\lambda}$ of the second problem satisfy the following secular equation

$$1 + c^T(A - \bar{\lambda}I)^{-1}c = 0. \tag{6}$$

2.3 Constraint quadratic optimization

Let A be a symmetric positive definite matrix of order N and c a given N vector. The quadratic optimization problem is stated as the following:

$$\min_x \quad x^T Ax - 2c^T x \tag{7}$$

with the constraint

$$x^T x = \alpha^2, \tag{8}$$

where α is a given scalar. Now let

$$\phi(x, \lambda) = x^T Ax - 2c^T x + \lambda(x^T x - \alpha^2) \tag{9}$$

where λ is the Lagrange multiplier. Differentiating (9) with respect to x, we are led to the equation

$$(A + \lambda I)x - c = 0$$

By the constraint (8), we are led to the problem of determining $\lambda > 0$ such that

$$c^T(A + \lambda I)^{-2}c = \alpha^2. \tag{10}$$

Furthermore, one can show the existence of a unique positive λ^* for which the above equation is satisfied. The solution of the original problem (7) and (8) is then $x^* = (A + \lambda^* I)^{-1}c$.

2.4 Trace and determinant

The trace and determinant problems are simply to estimate the quantities

$$\mathrm{tr}(A^{-1}) = \sum_{i=1}^{n} \epsilon_i^T A^{-1} \epsilon_i$$

and

$$\det(A)$$

for a given matrix A. For the determinant problem, it can be easily verified that for a symmetric positive definite matrix A:

$$\ln(\det(A)) = \mathrm{tr}(\ln(A)) = \sum_{i=1}^{n} e_i^T (\ln(A)) e_i. \tag{11}$$

Therefore, the problem of estimating the determinant is essentially to estimate the trace of the matrix natural logarithm function $\ln(A)$.

2.5 Partial eigenvalue sum

The partial eigenvalue sum problem is to compute the sum of all eigenvalues less than a prescribed value α of the generalized eigenvalue problem

$$Ax = \lambda Bx, \tag{12}$$

where A and B are real $N \times N$ symmetric matrices with B positive definite. Specifically, let $\{\lambda_i\}$ be the eigenvalues; one wants to compute the quantity

$$\tau_\alpha = \sum_{\lambda_i < \alpha} \lambda_i$$

for a given scalar α.

Let $B = LL^T$ be Cholesky decomposition of B, the problem (12) is then equivalent to

$$(L^{-1} A L^{-T}) L^T x = \lambda L^T x.$$

Therefore the partial eigenvalue sum of the matrix pair (A, B) is equal to the partial eigenvalue sum of the matrix $L^{-1} A L^{-T}$, which, in practice, does not need to be formed explicitly.

A number of approaches might be found in literature to solve such problem. Our approach will based on constructing a function f such that the trace of $f(L^{-1} A L^{-T})$ approximates the desired sum τ_α. Specifically, one wants to construct a function f such that

$$f(\lambda_i) = \begin{cases} \lambda_i, & \text{if } \lambda_i < \alpha \\ 0, & \text{if } \lambda_i > \alpha, \end{cases} \tag{13}$$

for $i = 1, 2, \ldots, N$. Then $\mathrm{tr}(f(L^{-1}AL^{-T}))$ is the desired sum τ_α. One of choices is to have the f of the form

$$f(\zeta) = \zeta g(\zeta) \tag{14}$$

where

$$g(\zeta) = \frac{1}{1 + \exp\left(\frac{\zeta - \alpha}{\kappa}\right)},$$

where κ is a constant. This function, among other names, is known as the Fermi-Dirac distribution function [15, p. 347]. In the context of a physical system, the usage of this distribution function is motivated by thermodynamics. It directly represents thermal occupancy of electronic states. κ is proportional to the temperature of the system, and α is the chemical potential (the highest energy for occupied states).

It is easily seen that $0 < g(\zeta) < 1$ for all ζ with horizontal asymptotes 0 and 1. $(\alpha, \frac{1}{2})$ is the inflection point of g and the sign of κ determines whether g is decreasing ($\kappa > 0$) or increasing ($\kappa < 0$). For our application, we want the sum of all eigenvalues less than α, so we use $\kappa > 0$. The magnitude of κ determines how "close" the function g maps $\zeta < \alpha$ to 1 and $\zeta > \alpha$ to 0. As $\kappa \to 0^+$, the function $g(\zeta)$ rapidly converges to the step function $h(\zeta)$.

$$h(\zeta) = \begin{cases} 1 & \text{if } \zeta < \alpha \\ 0 & \text{if } \zeta > \alpha. \end{cases}$$

The graphs of the function $g(\zeta)$ for $\alpha = 0$ and different values of the parameter κ are plotted in Figure 1.

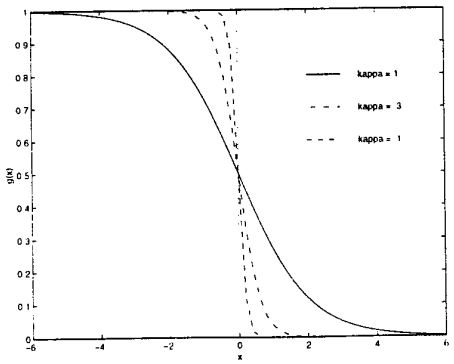


Fig. 1. Graphs of $g(\zeta)$ for different values of κ where $\alpha = 0$.

With this choice of $f(\zeta)$, we have

$$\tau_\alpha = \sum_{\lambda_i < \alpha} \lambda_i \approx \mathrm{tr}(f(L^{-1}AL^{-T})) = \sum_{i=1}^{n} e_i^T f(L^{-1}AL^{-T}) e_i. \tag{15}$$

In summary, the problem of computing partial eigenvalue sum becomes computing the trace of $f(L^{-1}AL^{-T})$.

3 Quadratic Form Computing

As we have seen, all those unusual eigenvalue problems presented in section 2 can be summarized as the problem of computing the quadratic form $u^T f(A)u$, where A is a $N \times N$ real matrix, and u is a vector, and f is a proper defined function. One needs to find an approximate of the quantity $u^T f(A)u$, or give a lower bound ℓ and/or an upper bound ν of it. Without loss of generality, one may assume $u^T u = 1$.

The quadratic form computing problem is first proposed in [6] for bounding the error of CG method for solving linear system of equations. It has been further developed in [11, 12, 2] and extended to other applications. The main idea is to first transform the problem of the quadratic form computing to a Riemann-Stieltjes integral problem, and then use Gauss-type quadrature rules to approximate the integral, which then brings the orthogonal polynomial theory and the underlying Lanczos procedure into the picture.

Let us go through the main idea. Since A is symmetric, the eigen-decomposition of A is given by $A = Q^T \Lambda Q$, where Q is an orthogonal matrix and Λ is a diagonal matrix with increasingly ordered diagonal elements λ_i. Then we have

$$u^T f(A)u = u^T Q^T f(\Lambda) Q u = \tilde{u}^T f(\Lambda)\tilde{u} = \sum_{i=1}^{N} f(\lambda_i)\tilde{u}_i^2,$$

where $\tilde{u} = (\tilde{u}_i) \equiv Qu$. The last sum can be considered as a Riemann-Stieltjes integral

$$u^T f(A)u = \int_a^b f(\lambda) d\mu(\lambda),$$

where the measure $\mu(\lambda)$ is a piecewise constant function and defined by

$$\mu(\lambda) = \begin{cases} 0, & \text{if } \lambda < a \le \lambda_1, \\ \sum_{j=1}^{i} \tilde{u}_j^2, & \text{if } \lambda_i \le \lambda < \lambda_{i+1} \\ \sum_{j=1}^{N} \tilde{u}_j^2 = 1, & \text{if } b \le \lambda_N \le \lambda \end{cases}$$

and a and b are the lower and upper bounds of the eigenvalues λ_i.

To obtain an estimate for the Riemann-Stieltjes integral, one can use the Gauss-type quadrature rule [9, 7]. The general quadrature formula is of the form

$$I[f] = \sum_{j=1}^{n} \omega_j f(\theta_j) + \sum_{k=1}^{m} \rho_k f(\tau_k). \tag{16}$$

where the weights $\{\omega_j\}$ and $\{\rho_k\}$ and the nodes $\{\theta_j\}$ are unknown and to be determined. The nodes $\{\tau_k\}$ are prescribed. If $m = 0$, then it is the well-known

Gauss rule. If $m = 1$ and $\tau_1 = a$ or $\tau_1 = b$, it is the Gauss-Radau rule. The Gauss-Lobatto rule is for $m = 2$ and $\tau_1 = a$ and $\tau_2 = b$.

The accuracy of the Gauss-type quadrature rules may be obtained by an estimation of the remainder $R[f]$:

$$R[f] = \int_a^b f(\lambda) d\mu(\lambda) - I[f].$$

For example, for the Gauss quadrature rule,

$$R[f] = \frac{f^{(2n)}(\eta)}{(2n)!} \int_a^b \left[\prod_{i=1}^{n} (\lambda - \theta_i) \right]^2 d\mu(\lambda),$$

where $a < \eta < b$. Similar formulas exist for Gauss-Radau and Gauss-Lobatto rules. If the sign of $R[f]$ is determined, then the quadrature formula $I[f]$ is a lower bound (if $R[f] > 0$) or an upper lower bound (if $R[f] < 0$) of the quantity $u^T f(A) u$.

Let us briefly recall how the weights and the nodes in the quadrature formula are obtained. First, we know that a sequence of polynomials $p_0(\lambda), p_1(\lambda), p_2(\lambda), \ldots$ can be defined such that they are orthonormal with respect to the measure $\mu(\lambda)$:

$$\int_a^b p_i(\lambda) p_j(\lambda) d\mu(\lambda) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

where it is assumed that the normalization condition $\int d\mu = 1$ (i.e., $u^T u = 1$). The sequence of orthonormal polynomials $\pi_j(\lambda)$ satisfies a three-term recurrence

$$\gamma_j p_j(\lambda) = (\lambda - \alpha_j) p_{j-1}(\lambda) - \gamma_{j-1} p_{j-2}(\lambda),$$

for $j = 1, 2, \ldots, n$ with $p_{-1}(\lambda) \equiv 0$ and $p_0(\lambda) \equiv 1$. Writing the recurrence in matrix form, we have

$$\lambda p(\lambda) = T_n p(\lambda) + \gamma_n p_n(\lambda) e_n$$

where

$$p(\lambda)^T = [p_0(\lambda), p_1(\lambda), \ldots, p_{n-1}(\lambda)], \qquad e_n^T = [0, 0, \ldots, 1]$$

and

$$T_n = \begin{pmatrix} \alpha_1 & \gamma_1 & & & \\ \gamma_1 & \alpha_2 & \gamma_2 & & \\ & \gamma_2 & \alpha_3 & \ddots & \\ & & \ddots & \ddots & \ddots \\ & & & \ddots & \gamma_{n-1} \\ & & & \gamma_{n-1} & \alpha_n \end{pmatrix}.$$

Then for the Gauss quadrature rule, the eigenvalues of T_n (which are the zeros of $p_n(\lambda)$) are the nodes θ_j. The weights ω_j are the squares of the first elements of the normalized (i.e., unit norm) eigenvectors of T_n.

For the Gauss-Radau and Gauss-Lobatto rules, the nodes $\{\theta_j\}$, $\{\tau_k\}$ and weights $\{\omega_j\}, \{\rho_j\}$ come from eigenvalues and the squares of the first elements of the normalized eigenvectors of an adjusted tridiagonal matrices of $\tilde{T}_{n+1}$, which has the prescribed eigenvalues a and/or b.

To this end, we recall that the classical Lanczos procedure is an elegant way to compute the orthonormal polynomials $\{p_j(\lambda)\}$ [16, 11]. We have the following algorithm in summary form. We refer it as the Gauss-Lanczos (GL) algorithm.

GL algorithm: Let A be a $N \times N$ real symmetric matrix, u a real N vector with $u^T u = 1$. f is a given smooth function. Then the following algorithm computes an estimation I_n of the quantity $u^T f(A) u$ by using the Gauss rule with n nodes.

- Let $x_0 = u$, and $x_{-1} = 0$ and $\gamma_0 = 0$
- For $j = 1, 2, \ldots, n$,
  1. $\alpha_j = x_{j-1}^T A x_{j-1}$
  2. $v_j = A x_{j-1} - \alpha_j x_{j-1} - \gamma_{j-1} x_{j-2}$
  3. $\gamma_j = \|v_j\|_2$
  4. $x_j = r_j / \gamma_j$
- Compute eigenvalues θ_k and the first elements ω_k of eigenvectors of T_n
- Compute $I_n = \sum_{k=1}^{n} \omega_k^2 f(\theta_k)$

We note that the "For" loop in the above algorithm is an iteration step of the standard symmetric Lanczos procedure [16]. The matrix A in question is only referenced here in the form of the matrix-vector product. The Lanczos procedure can be implemented with only 3 n-vectors in the fast memory. This is the major storage requirement for the algorithm and is an attractive feature for large scale problems.

On the return of the algorithm, from the expression of $R[f]$, we may estimate the error of the approximation I_n. For example, if $f^{(2n)}(\eta) > 0$ for any n and η, $a < \eta < b$, then I_n is a lower bound ℓ of the quantity $u^T f(A) u$.

Gauss-Radau-Lanczos (GRL) algorithm: To implement the Gauss-Radau rule with the prescribed node $\tau_1 = a$ or $\tau_1 = b$, the above GL algorithm just needs to be slightly modified. For example, with $\tau_1 = a$, we need to extend the matrix T_n to

$$\tilde{T}_{n+1} = \begin{bmatrix} T_n & \gamma_n e_n \\ \gamma_n e_n^T & \phi \end{bmatrix}.$$

Here the parameter ϕ is chosen such that $\tau_1 = a$ is an eigenvalue of $\tilde{T}_{n+1}$. From [10], it is known that

$$\phi = a + \delta_n,$$

where δ_n is the last component of the solution δ of the tridiagonal system

$$(T_n - aI)\delta = \gamma_n^2 e_n.$$

Then the eigenvalues and the first components of eigenvectors of $\tilde{T}_{n+1}$ gives the nodes and weight of the Gauss-Radau rule to compute an estimation $\tilde{I}_n$ of $u^T f(A)u$.

Furthermore, if $f^{(2n+1)}(\eta) < 0$ for any n and η, $a < \eta < b$, then $\tilde{I}_n$ (with b as a prescribed eigenvalue of $\tilde{T}_{n+1}$) is a lower bound ℓ of the quantity $u^T f(A)u$. $\tilde{I}_n$ (with a as a prescribed eigenvalue of $\tilde{T}_{n+1}$) is an upper bound ν.

Gauss-Lobatto-Lanczos (GLL) algorithm: To implement the Gauss-Lobatto rule, T_n computed in the GL algorithm is updated to

$$\hat{T}_{n+1} = \begin{bmatrix} T_n & \psi e_n \\ \psi e_n^T & \phi \end{bmatrix}.$$

Here the parameters ϕ and ψ are chosen so that a and b are eigenvalues of $\hat{T}_{n+1}$. Again, from [10], it is known that

$$\phi = \frac{\delta_n b - \mu_n a}{\delta_n - \mu_n} \qquad \text{and} \qquad \psi^2 = \frac{b+a}{\delta_n - \mu_n},$$

where δ_n and μ_n are the last components of the solutions δ and μ of the tridiagonal systems

$$(T_n - aI)\delta = \epsilon_n \qquad \text{and} \qquad (T_n - bI)\mu = \epsilon_n.$$

The eigenvalues and the first components of eigenvectors of $\hat{T}_{n+1}$ gives the nodes and weight of the Gauss-Lobatto rule to compute an estimation $\hat{I}_n$ of $u^T f(A)u$. Moreover, if $f^{(2n)}(\eta) > 0$ for any η, $a < \eta < b$, then $\hat{I}_n$ is an upper bound ν of the quantity $u^T f(A)u$.

Finally, we note that we need not always compute the eigenvalues and the first components of eigenvectors of the tridiagonal matrix T_n or its modifications $\tilde{T}_{n+1}$ or $\hat{T}_{n+1}$ for obtaining the estimation I_n or $\tilde{I}_n$, $\hat{I}_n$. We have following proposition.

Proposition 1. *For Gaussian rule:*

$$I_n = \sum_{k=1}^{n} \omega_k^2 f(\theta_k) = \epsilon_1^T f(T_n)\epsilon_1. \tag{17}$$

For Gauss-Radau rule:

$$\tilde{I}_n = \sum_{k=1}^{n} \omega_k^2 f(\theta_k) + \rho_1 f(\tau_1) = \epsilon_1^T f(\tilde{T}_{n+1})\epsilon_1. \tag{18}$$

For Gauss-Lobatto rule:

$$\hat{I}_n = \sum_{k=1}^{n} \omega_k^2 f(\theta_k) + \rho_1 f(\tau_1) + \rho_2 f(\tau_2) = \epsilon_1^T f(\hat{T}_{n+1})\epsilon_1. \tag{19}$$

Therefore, if the (1,1) entry of $f(T_n)$, $f(\tilde{T}_{n+1})$ or $f(\hat{T}_{n+1})$ can be easily computed, for example, $f(\lambda) = 1/\lambda$, we do not need to compute the eigenvalues and eigenvectors.

4 Solving the UEPs by Quadratic Form Computing

In this section, we use the GL, GRL and GLL algorithms for solving those unusual eigenvalue problems discussed in section 2.

Constraint eigenvalue problem Using the GL algorithm with the matrix A and the vector c, we have

$$c_1^T(A-\lambda I)^{-1}c = e_1^T(T_n-\lambda I)^{-1}e_1 + R,$$

where R is the remainder. Now we may solve reduced-order secular equation

$$e_1^T(T_n-\lambda I)^{-1}e_1 = 0$$

to find the largest λ as the approximate solution of the problem. This secular equation can be solved using the method discussed in [17] and its implementation available in LAPACK [1].

Modified eigenvalue problem Again, using the GL algorithm with the matrix A and the vector c, we have

$$1 + c^T(A-\bar{\lambda}I)^{-1}c = 1 + e_1^T(T_n-\bar{\lambda}I)^{-1}e_1 + R,$$

where R is the remainder. Then we may solve the eigenvalue problem of T_n to approximate some eigenvalues of A, and then solve reduced-order secular equation

$$1 + e_1^T(T_n-\bar{\lambda}I)^{-1}e_1 = 0$$

for $\bar{\lambda}$ to find some approximate eigenvalues of the modified eigenvalue problem.

Constraint quadratic programming By using the GRL algorithm with the prescribed node $\tau_1 = b$ for the matrix A and vector c, it can be shown that

$$c^T(A+\lambda I)^{-2}c \geq e_1^T(\tilde{T}_{n+1}+\lambda I)^{-2}e_1$$

for all $\lambda > 0$. Then by solving the reduced-order secular equation

$$e_1^T(\tilde{T}_{n+1}+\lambda I)^{-2}e_1 = \alpha^2$$

for λ, we obtain $\underline{\lambda}_n$, which is a lower bound of the solution λ^*: $\underline{\lambda}_n \leq \lambda^*$

On the other hand, using the GRL algorithm with the prescribed node $\tau_1 = a$, we have

$$c^T(A+\lambda I)^{-2}c \leq e_1^T(\tilde{T}_{n+1}+\lambda I)^{-2}e_1$$

for all $\lambda > 0$. Then by solving the reduced-order secular equation

$$e_1^T(\tilde{T}_{n+1}+\lambda I)^{-2}e_1 = \alpha^2$$

for λ. We have an upper bound $\bar{\lambda}_n$ of the solution λ^*: $\bar{\lambda}_n \geq \lambda^*$.

Using such two-sided approximation as illustrated in Figure 2, the iteration can be adaptively proceeded until the estimations $\underline{\lambda}_n$ and $\bar{\lambda}_n$ are sufficiently close, we then obtain an approximation

$$\lambda^* \approx \frac{1}{2}(\underline{\lambda}_n + \bar{\lambda}_n)$$

of the desired solution λ^*.

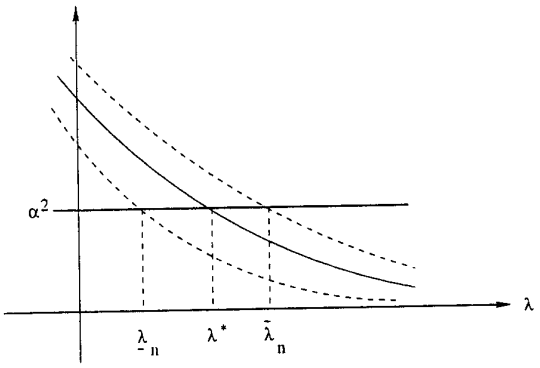


Fig. 2. Two-sided approximation approximation of the solution λ^* for the constraint quadratic programming problem (7) and (8).

Trace, determinant and partial eigenvalue sum As shown in sections 2.4 and 2.5, the problems of computing trace of the inverse of a matrix A, determinant of a matrix A and partial eigenvalue sum of a symmetric positive definite pair (A, B) can be summarized as the problem of computing the trace of a corresponding matrix function $f(H)$, where $H = A$ or $H = L^{-1}AL^{-T}$ and $f(\lambda) = 1/\lambda$, $\ln(\lambda)$ or $\lambda/\left(1+\exp(\frac{\lambda-\alpha}{\kappa})\right)$. To efficiently compute the trace of $f(H)$, instead of applying GR algorithm or its variations N times for each diagonal element of $f(H)$, we may use a Monte Carlo approach which only applies the GR algorithm m times to obtain an unbiased estimation of $\mathrm{tr}(f(H))$. For practical purposes, m can be chosen much smaller than N. The saving in computational costs could be significant. Such a Monte Carlo approach is based on the following lemma due to Hutchinson [14].

Proposition 2. *Let $C = (c_{ij})$ be an $N \times N$ symmetric matrix with $tr(C) \neq 0$. Let $\mathcal{V}$ be the discrete random variable which takes the values 1 and -1 each with probability 0.5 and let z be a vector of n independent samples from $\mathcal{V}$. Then $z^T C z$ is an unbiased estimator of $tr(C)$, i.e.,*

$$E(z^T C z) = tr(C),$$

and

$$var(z^T C z) = 2 \sum_{i \neq j} c_{ij}^2.$$

To use the above proposition in practice, one takes m such sample vectors z_i, and then uses GR algorithm or its variations to obtain an estimation $I_n^{(i)}$, a lower bound $\ell_n^{(i)}$ and/or an upper bound $\nu_n^{(i)}$ of the quantity $z_i^T f(H) z_i$:

$$\ell_n^{(i)} \leq z_i^T f(H) z_i \leq \nu_n^{(i)}.$$

Then by taking the mean of the m computed estimation $I_n^{(i)}$ or lower and upper bounds $\ell_n^{(i)}$ and $\nu_n^{(i)}$, we have

$$\mathrm{tr}(f(H)) \approx \frac{1}{m} \sum_{i=1}^{m} I_n^{(i)}$$

or

$$\frac{1}{m} \sum_{i=1}^{m} \ell_n^{(i)} \leq \frac{1}{m} \sum_{i=1}^{m} z_i^T f(H) z_i \leq \frac{1}{m} \sum_{i=1}^{m} \nu_n^{(i)}.$$

It is natural to expect that with a suitable sample size m, the mean of the computed bounds yields a satisfactory estimation of the quantity $\mathrm{tr}(f(H))$. To assess the quality of such estimation, one can also obtain probabilistic bounds of the approximate value [2].

5 Numerical Examples

In this section, we present some numerical examples to illustrate our quadratic form based algorithms for solving some of the unusual eigenvalue problems discussed in section 2.

5.1 Trace and determinant

Numerical results for a set of test matrices presented in Tables 1 and 2 are first reported in [2]. Some of these test matrices are model problems and some are from practical applications. For example, VFH matrix is from the analysis of transverse vibration of a Vicsek fractal. These numerical experiments are carried out on an Sun Sparc workstation. The so-called "exact" value is computed by using the standard methods for dense matrices. The numbers in the "Iter"-column are the number of iterations n required for the estimation $I_n^{(i)}$ to reach stationary value within the given tolerance value $tol = 10^{-4}$, namely,

$$|I_n - I_{n-1}| \leq tol * |I_n|.$$

The number of random sample vector z_i is $m = 20$. For those test matrices, the relative accuracy of the new approach within 0.3% to 8.2% may be sufficient for practical purposes.

Table 1. Numerical Results of estimating $\mathrm{tr}(A^{-1})$

Matrix	N	"Exact"	Iter	Estimated	Rel.err
Poisson	900	$5.126e+02$	30–50	$5.020e+02$	2.0%
VFH	625	$5.383e+02$	12–21	$5.366e+02$	0.3%
Wathen	481	$2.681e+01$	33–58	$2.667e+01$	0.5%
Lehmer	200	$2.000e+04$	38–70	$2.017e+04$	0.8%

Table 2. Numerical results of estimating $\ln(\det(A)) = \mathrm{tr}(\ln A)$

Matrix	N	"Exact"	Iter	Estimated	Rel.err
Poisson	900	$1.065e+03$	11–29	$1.060e+03$	0.4%
VFH	625	$3.677e+02$	10–14	$3.661e+02$	0.4%
Heat Flow	900	$5.643e+01$	4	$5.669e+01$	0.4%
Pei	300	$5.707e+00$	2–3	$5.240e+00$	8.2%

5.2 Partial eigenvalue sum

Here we present a numerical example from the computation of the total energy of an electronic structure. Total energy calculation of a solid state system is necessary in simulating real materials of technological importance [18]. Figure 3 shows a carbon cluster that forms part of a "knee" structure connecting nanotubes of different diameters and the distribution of eigenvalues such carbon structure with 240 atoms. One is interested in computing the sum of all these eigenvalues less than zero. Comparing the performance of our method with dense methods, namely symmetric QR algorithm and bisection method in LAPACK, our method achieved up to a factor of 20 speedup for large system on an Convex Exemplar SPP-1200 (see Table 3). Because of large memory requirements, we were not able to use LAPACK divide-and-conquer symmetric eigenroutines. Furthermore, algorithms for solving large-sparse eigenvalue problems, such as Lanczos method or implicitly restarted methods for computing some eigenvalues are found inadequate due to large number of eigenvalues required. Since the problem is required to be solved repeatly, we are now able to solve previously intractable large scale problems. The relative accuracy of new approach within 0.4% to 1.5% is satisfactory for the application [3].

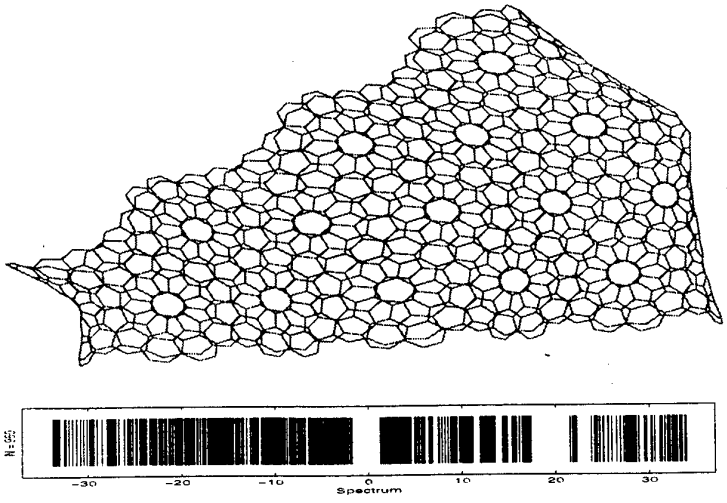


Fig. 3. A carbon cluster that forms part of a "knee" structure, and the corresponding spectrum

Table 3. Performance of our method vs. dense methods on Convex Exemplar SPP-1200. Here, 10 Monte Carlo samples were used to obtain estimates for each systems size.

		Dense methods			GR Algorithm		% Relative
n	m	Partial Sum	QR Time	BI Time	Estimate	Time	Error
480	349	-4849.8	7.4	7.6	-4850.2	2.8	0.01
960	648	-9497.6	61.9	51.8	-9569.6	18.5	0.7
1000	675	-9893.3	80.1	58.6	-10114.1	22.4	2.2
1500	987	-14733.1	253.6	185.6	-14791.8	46.4	0.4
1920	1249	-18798.5	548.3	387.7	-19070.8	72.6	1.4
2000	1299	-19572.9	616.9	431.8	-19434.7	78.5	0.7
2500	1660	-24607.6	1182.2	844.6	-24739.6	117.2	0.5
3000	1976	-29471.3	1966.4	1499.7	-29750.9	143.5	0.9
3500	2276	-34259.5	3205.9	2317.4	-33738.5	294.0	1.5
4000	2571	-39028.9	4944.3	3553.2	-39318.0	306.0	0.7
4244	2701	-41299.2	5915.4	4188.0	-41389.8	339.8	0.2

6 Concluding Remarks

In this paper, we have surveyed numerical techniques based on computing quadratic forms for solving some unusual eigenvalue problems. Although there exist some numerical methods for solving these problems (see [13] and references therein), most of these can be applied only for small and/or dense problems. The techniques presented here reference the matrix in question only through a matrix-vector product operation. Hence, they are more suitable for large sparse problems.

The new approach deserves further study; in particular, for error estimation and convergence of the methods. An extensive comparative study of the trade-offs in accuracy and computational costs between the new approach and other existing methods should be conducted.

Acknowledgement Z. B. was supported in part by an NSF grant ASC-9313958, an DOE grant DE-FG03-94ER25219.

References

1. Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., Sorensen, D.: LAPACK Users' Guide (second edition), SIAM, Philadelphia, 1995.
2. Bai, Z., Fahey, M., Golub, G.: Some large-scale matrix computation problems. J. Comp. Appl. Math. **74** (1996) 71–89.
3. Bai, Z., Fahey, M., Golub, G., Menon, M., Richter, E.: Computing partial eigenvalue sum in electronic structure calculations, Scientific Computing and Computational Mathematics Program, Computer Science Dept., Stanford University, SCCM-98-03, 1998.

4. Barrett. R., Berry. M., Chan. F., Demmel. J., Donato. J., Dongarra. J., Eijkhout. V., Pozo. R., Romine. C., van der Vorst., H.: Templates for the solution of linear systems: Building blocks for iterative methods. SIAM, Philadelphia, 1994.
5. Carey. C., Golub, G., Law, K.: A Lanczos-based method for structural dynamic reanalysis problems. Inter. J. Numer. Methods in Engineer., 37 (1994) 2857–2883.
6. Dahlquist, G., Eisenstat, S., Golub, G.: Bounds for the error of linear systems of equations using the theory of moments. J. Math. Anal. Appl. 37 (1972) 151–166.
7. Davis. P., Rabinowitz. P.: Methods of Numerical Integration. Academic Press, New York, 1984.
8. Demmel, J., Heath. M., van der Vorst., H.: Parallel linear algebra, in Acta Numerica, Vol.2, Cambridge Press, New York, 1993
9. Gautschi. W., A survey of Gauss-Christoffel quadrature formulae. In P. L Bultzer and F. Feher, editors, *E. B. Christoffel - the Influence of His Work on on Mathematics and the Physical Sciences*, pages 73–157. Birkhauser, Boston, 1981.
10. Golub, G.: Some modified matrix eigenvalue problems. SIAM Review, 15 (1973) 318–334.
11. Golub, G., Meurant, G.: Matrices, moments and quadrature, in Proceedings of the 15th Dundee Conference, June 1993, D. F. Griffiths and G. A. Watson. eds., Longman Scientific & Technical, 1994.
12. Golub, G., Strakoš, Z.: Estimates in quadratic formulas, Numerical Algorithms, 8 (1994) 241–268.
13. Golub. G., Van Loan. C.: Matrix Computations. Johns Hopkins University Press, Baltimore, MD, third edition, 1996.
14. Hutchinson, M.: A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines, Commun. Statist. Simula., 18 (1989) 1059–1076.
15. Kerstin. K., Dorman. K. R.: A Course in Statistical Thermodynamics, Academic Press, New York, 1971.
16. Lanczos, C.: An iteration method for the solution of the eigenvalue problem of linear differential and integral operators, J. Res. Natl. Bur. Stand. 45 (1950) 225–280.
17. Li., R.-C.: Solving secular equations stably and efficiently, Computer Science Division, Department of EECS, University of California at Berkeley, Technical Report UCB//CSD-94-851.1994
18. Menon. M., Richter. E., Subbaswamy. K. R.: Structural and vibrational properties of fullerenes and nanotubes in a nonorthogonal tight-binding scheme. J. Chem. Phys., 104 (1996) 5875–5882.

Parallel Preconditioners for Solving Nonsymmetric Linear Systems

Antonio J. García-Loureiro[1], Tomás F. Pena[1],
J.M. López-González[2], and Ll. Prat Viñas[2]

[1] Dept. Electronics and Computer Science. Univ. Santiago de Compostela
Campus Sur. 15706 Santiago de Compostela. Spain.
antonio@dec.usc.es, tomas@dec.usc.es
[2] Dept. of Electronic Engineering. Univ. Politécnica de Catalunya
Modulo 4. Campus Norte.c) Jordi Girona 1 y 3. 08034 Barcelona. Spain
jmlopezg@eel.upc.es

Abstract. *In this work we present a parallel version of two preconditioners. The first one, is based on a partially decoupled block form of the ILU. We call it Block-ILU(fill,τ,overlap), because it permits the control of both, the block fill and the block overlap. The second one, is based on the SPAI (SParse Approximate Inverse) method. Both methods are analysed and compared to the ILU preconditioner using the Bi-CGSTAB to solve general sparse, nonsymmetric systems. Results have been obtained for different matrices. The preconditioners have been compared in terms of robustness, speedup and time of execution, to determine which is the best one in each situation. These solvers have been implemented for distributed memory multicomputers, making use of the MPI message passing standard library.*

Keywords: parallel preconditioners, nonsymmetric linear systems, Block-ILU, SPAI, Bi-CGSTAB.

1 Introduction

In the development of simulation programs in different research fields, from fluid mechanics to semiconductor devices, the solution of the systems of equations which arise from the discretization of partial differential equations, is the most CPU consuming part [13]. In general, the matrices are very large, sparse, nonsymmetric and are not diagonal dominant [3, 12]. So, using an effective method to solve the system is essential.

We are going to consider a linear system of equations such as:

$$Ax = b, \qquad A \in \mathbb{R}^{n \times n}, \quad x, b \in \mathbb{R}^n \tag{1}$$

where A is a sparse, nonsymmetric matrix.

Direct methods, such as Gaussian elimination, LU factorization or Cholesky factorization may be excessively costly in terms of computational time and memory, specially when n is large. Due to these problems, iterative methods [1, 14] are generally preferred for the solution of large sparse systems. In this work we have chosen a non stationary iterative solver, the Bi–Conjugate Gradient Stabilized [19]. Bi–CGSTAB is one of the methods that obtains better results in the solution of non–symmetric linear systems, and its attractive convergence behaviour has been confirmed in many numerical experiments in different fields [7].

In order to reduce the number of iterations needed in the Bi–CGSTAB process, it is convenient to precondition the matrices. This is, transform the linear system into an equivalent one, in the sense that it has the same solution, but which has more favourable spectral properties.

Looking for efficient parallel preconditioners is a very important topic in current research in the field of scientific computing. A broad class of preconditioners are based on incomplete factorizations (incomplete Cholesky or ILU) of the coefficient matrix. One important problem associated with these preconditioners is their inherently sequential character. This implies that they are very hard to parallelise, and only a modest account of parallelism can be attained, with complicated implementations. So, it is important to find alternative forms of preconditioners that are more suitable for parallel architectures.

The first preconditioner we present is based on a partially decoupled block form of the ILU [2]. This new version, called Block–ILU(*fill*,τ,*overlap*), permits the control of its effectiveness through a dropping parameter τ and a block fill–in parameter. Moreover, it permits the control of the overlap between the blocks. We have verified that the fill–in control is very important for getting the most out of this preconditioner. Its main advantage is that it presents a very efficient parallel execution, because it avoids the data dependence of sequential ILU, obtaining high performance and scalability. As a disadvantage is that it is less robust than complete ILU, due to the loss of information, and this can be a problem in very bad conditioned systems.

The second preconditioner we present is an implementation of preconditioner SPAI (*SParse Approximate Inverse*) [5, 8]. This alternative has been proposed in the last few years as an alternative to ILU, in situations where the last obtain very poor results (situations which often arise when the matrices are indefinite or have large nonsymmetric parts). These methods are based on finding a matrix M which is a direct approximation to the inverse of A, so that $AM \approx I$.

This paper presents a parallel version of these preconditioners. Section 2 presents the iterative methods we have used. Section 3 introduces the characteristics of the Block–ILU and the SPAI preconditioners. Section 4 indicates the numerical experiment we have studied. The conclusions are given in Section 5.

2 Iterative Methods

The *iterative methods* are a wide range of techniques that use successive approximations to obtain more accurate solutions to linear systems at each step. There

are two types of iterative methods. Stationary methods, like Jacobi, Gauss–Seidel, SOR, etc., are older, simpler to understand and implement, but usually not very effective. Nonstationary methods, like Conjugate Gradient, Minimum Residual, QMR, Bi–CGSTAB, etc., are a relatively recent development and can be highly effective. These methods are based on the idea of sequences of orthogonal vectors.

In recent years the Conjugate Gradient–Squared (CGS) method [1] has been recognized as an attractive variant of the Bi–Conjugate Gradient (Bi–CG) for the solution of certain classes of nonsymmetric linear systems. Recent studies indicate that the method is often competitive with other well established methods, such as GMRES [15]. The CG–S method has tended to be used in the solution of two or tree–dimensional problems, despite its irregular convergence pattern, because when it works -which is most of the time- it works quite well. Recently, van der Vorst [19] has presented a new variant of Bi–CG, called Bi–CGSTAB, which combines the efficiency of CGS with the more regular convergence pattern of Bi–CG.

In this work we have chosen the Bi–Conjugate Gradient Stabilized [1, 19], because of its attractive convergence behaviour. This method was developed to solve nonsymmetric linear systems while avoiding irregular convergence patterns of the Conjugate Gradient Squared methods. Bi–CGSTAB requires two matrix–vector products and four inner products per iteration.

In order to reduce the number of iterations needed in the Bi–CGSTAB process, it is convenient to precondition the matrices. The preconditioning can be applied in two ways: either we solve the explicitly preconditioned system using the normal algorithm, or we introduce the preconditioning process in the iterations of the Bi–CGSTAB. This last method is usually preferred.

3 Preconditioners

The rate at which an iterative method converges depends greatly on the spectrum of the coefficient matrix. Hence iterative methods usually involve a second matrix that transforms the coefficient matrix into one with a more favorable spectrum. A preconditioner is a matrix that affects such a transformation.

We are going to consider a linear system of equations such as:

$$Ax = b, \qquad A \in \mathbb{R}^{n \times n}, \quad x, b \in \mathbb{R}^n \tag{2}$$

where A is a large, sparse, nonsymmetric matrix.

If a matrix M is right–approximates coefficient matrix A in some way, we can transform the original system as follows:

$$Ax = b \quad \rightarrow \quad AM^{-1}(Mx) = b \tag{3}$$

Similarly, a left–approximates can be defined by:

$$Ax = b \quad \rightarrow \quad M^{-1}Ax = M^{-1}b \tag{4}$$

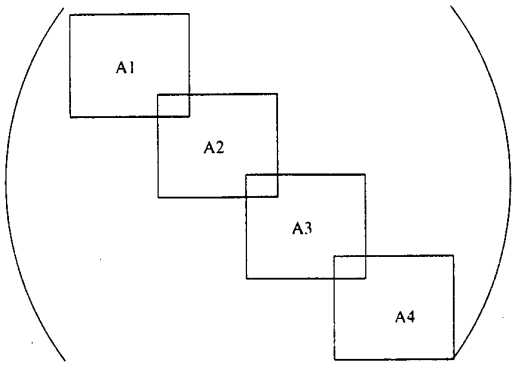


Fig. 1. Matrix splits in blocks

Another way of deriving the preconditioned conjugate gradients method would be to split the preconditioner as $M = M_1 M_2$, where the matrices M_1 and M_2 are called the left and right preconditioners, and to transform the system as

$$Ax = b \quad \rightarrow \quad M_1^{-1} A M_2^{-1} (M_2 x) = M_1^{-1} b \tag{5}$$

In this section we present a parallel version of two preconditioners. The first one, is based on a partially decoupled block form of the ILU. We call it Block–ILU(fill,τ,overlap), because it permits the control of both the block fill and the block overlap. The second one is based on the SPAI (SParse Approximate Inverse) method. Both methods are analysed and compared to the ILU preconditioner using the Bi–CGSTAB to solve general sparse, nonsymmetric systems.

3.1 Parallel Block–ILU preconditioner

In this section we present a new version of a preconditioner based on a partially decoupled block form of the ILU [2]. This new version, called Block–ILU(*fill*,τ,*overlap*), permits the control of its effectiveness through a dropping parameter τ and a block fill–in parameter. Moreover, it permits the control of the overlap between the blocks. We have verified that the fill–in control is very important for getting the most out of this preconditioner. The original matrix is subdivided into a number of overlapping blocks, and each block is assigned to a processor. This setup produces a partitioning effect represented in Figure 1, for the case of 4 processors, where the ILU factorization for all the blocks is computed in parallel, obtaining $A_i = L_i U_i$, $1 \leqslant i \leqslant p$, where p is the number of blocks. Due to the characteristics of this preconditioner, there is a certain loss of information. This means that the number of iterations will increase as the number of blocks increases (as a direct consequence of increasing the number of processors). This loss can be compensated to a certain extent by the information provided by the overlapping zones.

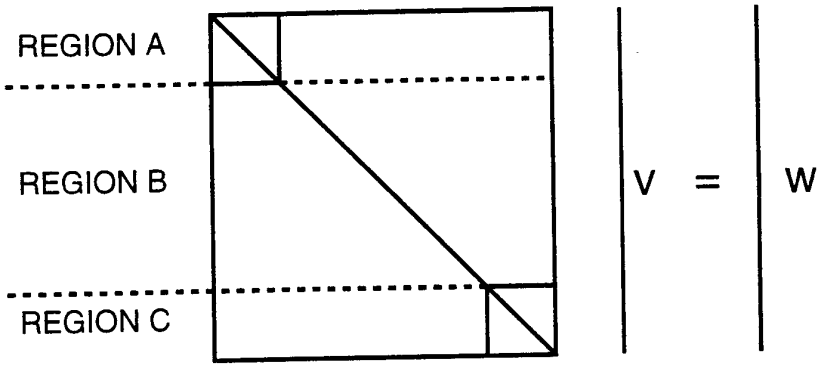


Fig. 2. Scheme of one block

To create the preconditioner the rows of each block indicated for the parameter *overlap* are interchanged between the processors. These rows correspond to regions A and C of figure 2. After, the factorization is carried out. Within the loop of algorithm of resolution it is necessary to carry out the operation of preconditioning

$$L_i U_i v = w \tag{6}$$

To reduce the number of operations of the algorithm, each processor only works with its local rows. The first operation is to extend vector w's information to the neighbouring processors. Later we carry out in each processor the resolution of the superior and inferior triangular system to calculate vector v. As regions A and C have also been calculated by other processors, the value that we obtain will vary in different processors. In order to avoid this and improve the convergency of the algorithm it is necessary to interchange these data and calculate the average of these values.

The main advantage of this method is that it presents a very efficient parallel execution, because it avoids the data dependence of sequential ILU, thereby obtaining high performance and scalability. A disadvantage is that it is less robust than complete ILU, due to the loss of information, and this can be a problem in very bad conditioned systems, as we will show in section 4.

3.2 Parallel SPAI preconditioner

One of the main drawback of ILU preconditioner is the low parallelism it implies. A natural way to achive parallelism is to compute an approximate inverse M of A, such that $M \cdot A \simeq I$ in some sense. A simple technique for finding approximate inverses of arbitrary sparse matrices is to attempt to find a sparse matrix M which minimizes the Frobenius norm of the residual matrix $AM - I$,

$$F(M) = \|AM - I\|_F^2 \tag{7}$$

A matrix M whose value $F(M)$ is small would be a right–approximate inverse of A. Similarly, a left–approximate inverse can be defined by using the objective function

$$\|MA - I\|_F^2 \tag{8}$$

These cases are very similar. The objective function 7 decouples into the sum of the squares of the 2–norms of the individual columns of the residual matrix $AM - I$,

$$F(M) = \|AM - I\|_F^2 = \sum_{j=1}^{n} \|Am_j - e_j\|_2^2 \tag{9}$$

in which e_j and m_j are the j–th columns of the identity matrix and of the matrix M. There are two different ways to proceed in order to minimize 9. The first one consists of in minimizing it globally as a function of the matrix M, e.g., by a gradient–type method. Alternatively, in the second way the individual functions

$$f_j(m) = \|Am_j - e_j\|_F^2, j = 1, \ldots, n \tag{10}$$

can be minimized. This second approach is attractive for parallel computers, and it is the one we have used in this paper. A good, inherently parallel solution would be to compute the columns k of M, m_k, in an independent way from each other, resulting:

$$\|AM - I\|_F^2 = \sum_{k=1}^{n} \|(AM - I)e_k\|_2^2 \tag{11}$$

The solution of 11 can be organized into n independent systems,

$$\min_{m_k} \|Am_k - e_k\|_2, \quad k = 1, ..., n, \quad e_k = (0, ..., 0, 1, 0, ..., 0)^T \tag{12}$$

We have to solve n systems of equations. If these linear systems were solved without taking advantage of sparsity, the cost of constructing the preconditioner would be of order n^2. This is because each of the n columns would require $O(n)$ operations. Such a cost would become unacceptable for large linear systems. To avoid this, the iterations must be performed in *sparse–sparse mode*. As A is sparse, we could work with systems of much lower dimension. Let $L(k)$ be the set of indices j such that $m_k(j) \neq 0$. We denote the reduced vector of unknowns as $m_k(L)$ by $\hat{m}_k(L)$ and the resulting submatrix $A(L, L)$ as $\hat{A}$. Similarly, we define $\hat{e}_k = e_k(L)$. Now, solving 12 is transformed into solving:

$$\min_{m_k} \|\hat{A}\hat{m}_k - \hat{e}_k\|_2 \tag{13}$$

Due to the sparsity of A and M, the dimension of systems 13 is very small. To solve these systems we have chosen direct methods. We are using these methods instead of an iterative one, mainly because the systems 13 are very small and almost dense. Of the different alternatives we have concentrated on QR and LU methods [17].

The QR factorization of matrix $A \in \mathbb{R}^{m \times n}$ is given by $A = QR$ where R is an m–by–n upper triangular matrix and Q is an m–by–m unitary matrix.

This factorization is better than LU because it can be used for the case of non squared matrices, and also works in some cases in which LU fails due to problems with too small pivots [10]. The cost of this factorization is $O(\frac{4}{3}n^3)$. The other direct method we have tested is LU. This factorization and the closely related Gaussian elimination algorithm are widely used in the solution of linear systems of equations. LU factorization expresses the coefficient matrix, A, as the product of a lower triangular matrix, L, and an upper triangular matrix, U. After factorization, the original system of equations can be written as a pair of triangular systems,

$$Ax = b \tag{14}$$

$$Ly = b \quad Ux = y \tag{15}$$

The first of the systems can be solved by forward reduction, and then back substitution can be used to solve the second system to give x. The advantage of this factorization is that its cost is $O(\frac{2}{3}n^3)$, lower than that of QR. We have implemented the two solvers in our code, specifically, the QR and the LU decomposition with pivoting. An efficient implementation consists of selecting the QR method if the matrix is not squared. In the case that it is squared, we will resolve the system by using LU, as this is faster than QR. Morever, there is also the possibility of using QR if some error is produced in the construction of the factorization LU.

In the next section we have compared the results we have obtained with these methods. In this code the SPAI parameter $\mathcal{L}$ indicates the number of neighbours of each point we use to reduce the system. The main drawback of preconditioners based on the SPAI idea is that they need more computations than the rest. So, in the simplest situations and when the number of processors is small, they may be slower than ILU based preconditioners.

4 Numerical experiments

4.1 Test problem specification

The matrices we have tested are from the simulation of heterojunction bipolar transistors [9, 11]. These matrices are highly sparse, not symmetric and, in general, not diagonal dominant. They were obtained by applying the method of finite elements to heterojunction bipolar devices, in concrete for transistors of InP/InGaAs [6].

The basic equations of the semiconductor devices are Poisson's eq. and electron and hole continuity, in a stationary state:

$$div(\varepsilon \nabla \psi) = q(p - n + N_D^+ - N_A^-) \tag{16}$$

$$div(J_n) = qR \tag{17}$$

$$div(J_p) = -qR \tag{18}$$

where ψ is the electrostatic potential, q is the electronic charge, ε is the dielectric constant of the material, n and p are the electron and hole densities, N_D^+ and

Table 1. Time (sec) for Block-ILU

Proc	2	4	6	8	10
FILL=0/Overlap=1	0.86	0.47	0.29	0.21	0.18
FILL=0/Overlap=3	0.83	0.41	0.29	0.22	0.18
FILL=0/Overlap=6	0.83	0.43	0.29	0.21	0.17
FILL=2/Overlap=1	0.38	0.18	0.12	0.094	0.077
FILL=4/Overlap=1	0.38	0.19	0.12	0.095	0.077

N_A^+ are the doping effective concentration and J_n and J_p are the electron and hole current densities, respectively. The term R represents the volume recombination term, taking into account Schokley–Read–Hall, Auger and band–to–band recombination mechanisms [20].

For this type of semiconductors it is usual to apply at first a Gummel type method of resolution [16], which uncouples the three equations and allows us to obtain an initial solution for the system coupled with the three equations. For the semiconductors we use we have to solve the three equations simultaneously. The pattern of these matrices is similar to those in other fields such as applications of CFD [3, 4].

We have distributed the matrix in rows and have obtained an optimum distribution of the work load among the processors.

All the results have been obtained in a CRAY T3E multicomputer [18]. We have programmed it using the SPMD paradigm, with the MPI library, and we have obtained results with several matrices of different characteristics.

4.2 Parallel Block–ILU preconditioner

We have carried out different tests to study how the parameters of *fill–in* and *overlap* affect the time of calculation and speedup for the resolution of a system of equations. In tables 1 and 2 we show the times of execution and speedup for a badly conditioned matrix with $N = 25000$. Time is measured from two processors onwards, because we have memory problems trying to run the code in a single processor. So the speedup is computed as:

$$speedup_p = \frac{T_p}{\frac{p}{2}T_2} \tag{19}$$

where T_p is the time of execution with two processors.

With respect to the results shown in table 1 note that, if we maintain constant the value of the *fill–in*, when the value of the *overlap* is increased the time of execution hardly varies. This is because the only variation is in the size of the message to be transmitted, whereas the size of the *overlap* zone in comparison to the total is minimum. Therefore the increase in the computations is small. However, if we maintain constant the value of the *overlap* and increase the *fill–in* a significant variation is observed. This is because the number of iterations decreases considerably

Table 2. Speedup for Block-ILU

Proc	2	4	6	8	10
FILL=0/Overlap=1	1.0	1.82	2.96	4.09	4.77
FILL=0/Overlap=3	1.0	2.02	2.86	3.77	4.66
FILL=0/Overlap=6	1.0	1.91	2.84	3.92	4.53
FILL=2/Overlap=1	1.0	2.10	3.16	4.04	4.93
FILL=4/Overlap=1	1.0	2.00	3.16	4.0	4.93

Table 3. Time (sec) for SPAI with LU

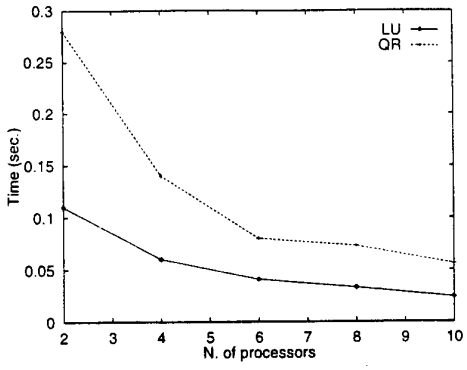
Proc	2	4	6	8	10	Iter.
$\mathcal{L}$=0	2.19	1.10	0.76	0.54	0.46	47
$\mathcal{L}$=1	1.93	0.97	0.66	0.48	0.41	35
$\mathcal{L}$=2	1.51	0.86	0.52	0.38	0.32	22
$\mathcal{L}$=3	1.41	0.73	0.49	0.35	0.32	17
$\mathcal{L}$=4	1.46	0.73	0.49	0.37	0.31	13
$\mathcal{L}$=5	1.60	0.79	0.51	0.39	0.33	11

As regards the values of speedup in table 2, the values obtained are significantly better in all cases, although the algorithm obtains slightly better results when the level of *fill–in* is increased for a constant level of *overlap*. However, for a constant level of *fill–in* the speedup decreases very smoothly as the level of overlap increases. This is because it is necessary to carry out a large number of operations and the cost of communications is also a little higher. From the results obtained it is possible to conclude that the best option is to choose the lowest value of overlap with which we can assure convergency with an average value of *fill–in*.

4.3 Parallel SPAI preconditioner

First we are going to compare the results we have obtained with the two direct solvers we have implemented in section 3.2. For a bad conditioned system of $N = 25000$ we have obtained the results shown in figure 3. These data refer to the cost of generating the matrix for each node with an overlap level 1. In this case resulting submatrices are of rank 3. Note that the cost of QR factorization is significantly higher than that of LU. This difference is much larger for higher values of the overlap level.

Table 3 shows the time used to solve a badly conditioned matrix with $N = 25000$, as well as the number of iterations of the Bi-CGSTAB solver. Note that as the value of parameter $\mathcal{L}$ increases, the number of iterations decreases because the preconditioner is more exact. As regards speedup, in all the cases values close to optimum are obtained. and in some cases even surpassed due to phenomena of superlineality. For this class of matrices the optimum value of parameter $\mathcal{L}$



(a) Time versus number of processors

Fig. 3. QR versus LU on the CRAY T3E with N=25000 (bad conditioned system)

would be 3 or 4. From the rest of results we can conclude that the more diagonally dominant the matrix, the smaller is the optimum value of this parameter, and inversely, for worse conditioned matrices we will need higher values of $\mathcal{L}$ to assure the convergency.

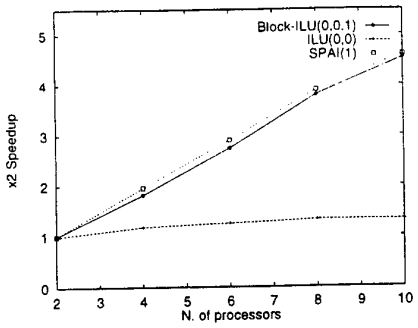
4.4 Parallel Block–ILU versus Parallel SPAI

In order to test the effectiveness of the parallel implementation of Block–ILU and SPAI, we have compared them to a parallel version of the ILU($fill$,τ) preconditioner.

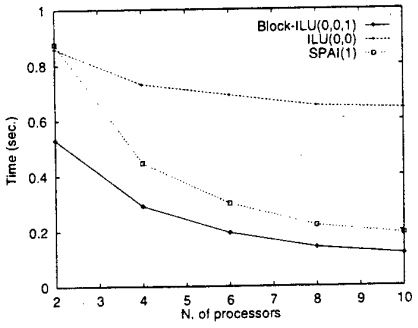
In Figure 4, results are shown for the complete solution of a system of equations with $N = 25000$, where matrix A is a well–conditioned one (diagonal dominant). Again time is measured from two processors, because we have memory problems trying to run the code in a single processor. It can be seen, in Figure 4(a), that the parallel SPAI method obtains the best speedup, and that parallel Block–ILU(0,0,1) obtains very similar results. However, the ILU(0,0) preconditioner obtains very bad results. This is because of the bottleneck implied in the solution of the upper and lower sparse triangular systems. On the other hand, parallel SPAI is slower (Figure 4(b)) when the number of processors is small, because of the high number of operations it implies.

In Figure 5, results are shown for a matrix with $N = 25000$, corresponding to a poorly–conditioned system. Again (Figure 5(a)) parallel SPAI and Block–ILU(0,0,1) obtain very similar speedup results. The ILU(0,0) preconditioner obtains the worst results. And again, parallel SPAI is the slower solution when the number of processors is small (Figure 5(b)).

From the point of view of scalability, parallel Block–ILU is worse than parallel SPAI. This is due to the fact that Block–ILU suffers a loss of information with



(a) Speedup versus number of processors

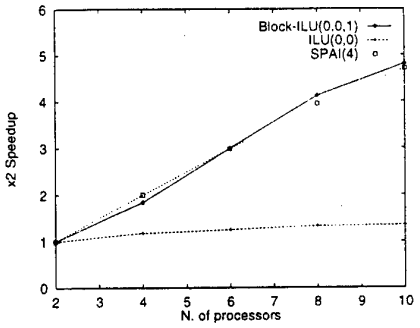


(b) Time versus number of processors

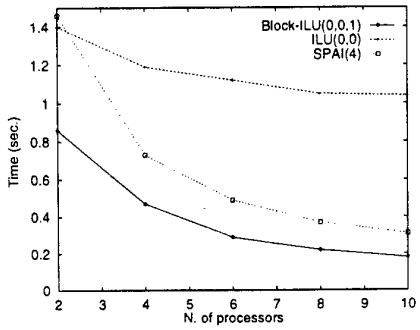
Fig. 4. Results on the CRAY T3E with $N = 25000$ (well conditioned system)

respect to the sequential algorithm when the number of processors increases. This means that, with some matrices, the number of iterations, and, therefore, the total time for the BI–CGSTAB to converge, grows when the number of processors increases, thereby degrading the effectiveness of the preconditioner.

Figure 6 shows the results for a non diagonally dominant and very badly conditioned matrix with $N = 25000$. In this case, the system converges with the three preconditioners, but a significant difference is noted between the SPAI and the incomplete factorizations. Note that with preconditioner SPAI we obtain a nearly ideal value of speedup, whereas in the other cases this hardly reaches 1. irrespective of the number of processores. However, if we examine the measures of time, it can be established that the fastest preconditioner is the ILU(3,0), together with the Block–ILU(3,0,3), although this time hardly varies with different numbers of processors. On the other hand, the SPAI is much slower than



(a) Speedup versus number of processors



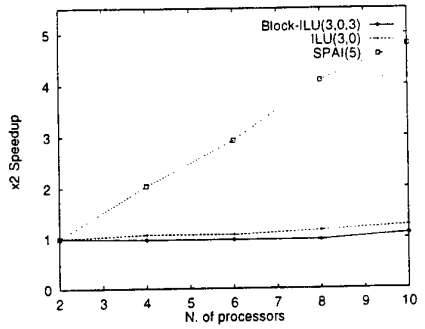
(b) Time versus number of processors

Fig. 5. Results on the CRAY T3E with $N = 25000$ (bad conditioned system)

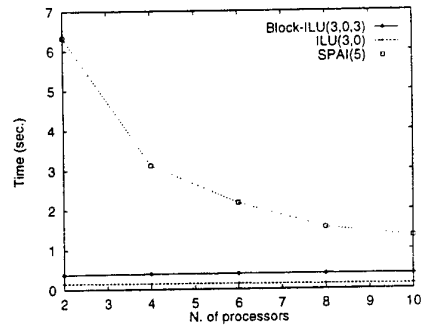
the other two. The motive for this behaviour is that, on the one hand, Block–ILU increases considerably the number of iterations as the number of processors is increased, due to the loss of information that this method implies. This increase compensates the reduction in the cost for iteration, which means that the speedup does not increase. On the other hand, to guarantee convergency we must use SPAI with high values of $\mathcal{L}$, which supposes a high cost of each iteration. However, the number of iterations does not grow as the number of processors increases, and thereby we obtain a high level of speedup. With a large number of processors, Parallel SPAI probably overcomes ILU based preconditioners.

5 Conclusions

Choosing the best preconditioner is going to be conditioned by the characteristics of the system we have to solve. When it is not a very badly conditioned



(a) Speedup versus number of processors



(b) Time versus number of processors

Fig. 6. Results on the CRAY T3E with $N = 25000$ (very bad conditioned system)

system, parallel Block–ILU appears to be the best solution, because of both the high level of speedup it achieves and the reduced time it requires to obtain the final solution. The Parallel SPAI preconditioner obtains very good results in scalability, so it could be the best choice when the number of processors grows. Moreover, we have verified that it achieves convergence in some situations where ILU based preconditioners fail. Finally, the direct parallel implementations of ILU obtain very poor results.

Acknowledgements

The work described in this paper was supported in part by the Ministry of Education and Science (CICYT) of Spain under projects TIC96–1125–C03 and TIC96–1058. We want to thank CIEMAT (Madrid) for providing us access to the Cray T3E multicomputer.

References

1. R. Barrett, M. Berry, et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1994.
2. G. Radicati di Brozolo and Y. Robert. Parallel Conjugate Gradient-like algorithms for solving sparse nonsymmetric linear systems on a vector multiprocessor. *Parallel Computing*, 11:223–239, 1989.
3. A. Chapman, Y. Saad, and L. Wigton. High order ILU preconditioners for CFD problems. Technical report, Minnesota Supercomputer Institute. Univ. of Minnesota, 1996.
4. Filomena D. d'Almeida and Paulo B. Vasconcelos. Preconditioners for nonsymmetric linear systems in domain decomposition applied to a coupled discretization of Navier-Stokes equations. In *Vector and Parallel Processing - VECPAR'96*, pages 295–312. Springer-Verlag, 1996.
5. V. Deshpande, M. Grote, P. Messmer, and W. Sawyer. Parallel implementation of a sparse approximate inverse preconditioner. In Springer-Verlag, editor, *Proceedings of Irregular'96*, pages 63–74, August 1996.
6. A.J. García-Loureiro, J.M. López-González, T. F. Pena, and Ll. Prat. Numerical analysis of abrupt heterojunction bipolar transistors. *International Journal of Numerical Modelling: Electronic Networks, Devices and Fields*, 1998. (in press).
7. A.J. García-Loureiro, T. F. Pena, J.M. López-González, and Ll. Prat. Preconditioners and nonstationary iterative methods for semiconductor device simulation. In *Conferencia de Dispositivos Electrónicos (CDE-97)*, pages 403–409. Universitat Politecnica de Catalunya, Barcelona, February 1997. in spanish.
8. Marcus J. Grote and Thomas Huckle. Parallel preconditioning with sparse approximate inverses. *Siam J. Sci. Comput.*, 18(3):838–853, May 1997.
9. K. Horio and H. Yanai. Numerical modeling of heterojunctions including the heterojunction interface. *IEEE Trans. on ED*, 37(4):1093–1098, April 1990.
10. David Kincaid and Ward Cheney. *Numerical Analysis.* Brooks/Cole, 1991.
11. J. M. Lopez-Gonzalez and Lluis Prat. Numerical modelling of abrupt InP/InGaAs HBTs. *Solid-St. Electron*, 39(4):523–527, 1996.
12. T.F. Pena, J.D. Bruguera, and E.L. Zapata. Finite element resolution of the 3D stationary semiconductor device equations on multiprocessors. *J. Integrated Computer-Aided Engineering*, 4(1):66–77, 1997.
13. C.S. Rafferty, M.R. Pinto, and R.W. Dutton. Iterative methods in semiconductors device simulation. *IEEE trans on Computer-Aided Design*, 4(4):462–471, October 1985.
14. Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Co., 1996.
15. Y. Saad and M.H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7:856–869. 1986.
16. D.L. Scharfetter and H.K. Gummel. Large-signal analysis of a silicon read diode oscillator. *IEEE Trans. on ED*, pages 64–77, 1969.
17. H. R. Schwarz. *Numerical Analysis.* John Wiley & Sons, 1989.
18. S. L. Scott. Synchronization and communication in the T3E multiprocessor. Technical report, Inc. Cray Research, 1996.
19. A. Van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 13:631–644, 1992.
20. C.M. Wolfe, N. Holonyak, and G.E. Stillman. *Physical Properties of Semiconductors*, chapter 8. Ed. Prentice Hall, 1989.

Parallel Preconditioned Solvers for Large Sparse Hermitian Eigenproblems

Achim Basermann

C&C Research Laboratories, NEC Europe Ltd.
Rathausallee 10, 53757 Sankt Augustin. Germany
basermann@ccrl-nece.technopark.gmd.de
http://www.ccrl-nece.technopark.gmd.de/~baserman/

Abstract. Parallel preconditioned solvers are presented to compute a few extreme eigenvalues and -vectors of large sparse Hermitian matrices based on the Jacobi-Davidson (JD) method by G.L.G. Sleijpen and H.A. van der Vorst. For preconditioning, an adaptive approach is applied using the QMR (Quasi-Minimal Residual) iteration. Special QMR versions have been developed for the real symmetric and the complex Hermitian case. To parallelize the solvers, matrix and vector partitioning is investigated with a data distribution and a communication scheme exploiting the sparsity of the matrix. Synchronization overhead is reduced by grouping inner products and norm computations within the QMR and the JD iteration. The efficiency of these strategies is demonstrated on the massively parallel systems NEC Cenju-3 and Cray T3E.

1 Introduction

The simulation of quantum chemistry and structural mechanics problems is a source of computationally challenging, large sparse real symmetric or complex Hermitian eigenvalue problems. For the solution of such problems, parallel preconditioned solvers are presented to determine a few eigenvalues and -vectors based on the Jacobi-Davidson (JD) method [9].

For preconditioning, an adaptive approach using the QMR (Quasi-Minimal Residual) iteration [2, 5, 7] is applied, i.e., the preconditioning system of linear equations within the JD iteration is solved iteratively and adaptively by checking the residual norm within the QMR iteration [3, 4]. Special QMR versions have been developed for the real symmetric and the complex Hermitian case.

The matrices A considered are *generalized sparse*, i.e., the computation of a matrix-vector multiplication $A \cdot v$ takes considerably less than n^2 operations. This covers ordinary sparse matrices as well as dense matrices from quantum chemistry built up additively from a diagonal matrix, a few outer products, and an FFT. In order to exploit the advantages of such structures with respect to operational complexity and memory requirements when solving systems of linear equations or eigenvalue problems, it is natural to apply iterative methods.

To parallelize the solvers, matrix and vector partitioning is investigated with a data distribution and a communication scheme exploiting the sparsity of the

matrix. Synchronization overhead is reduced by grouping inner products and norm computations within the QMR and the JD iteration. Moreover, in the complex Hermitian case, communication coupling of QMR's two independent matrix-vector multiplications is investigated.

2 Jacobi-Davidson Method

To solve large sparse Hermitian eigenvalue problems numerically, variants of a method proposed by Davidson [8] are frequently applied. These solvers use a succession of subspaces where the update of the subspace exploits approximate inverses of the problem matrix, A. For A, $A = A^H$ or $A^* = A^T$ holds where A^* denotes A with complex conjugate elements and $A^H = (A^T)^*$ (transposed and complex conjugate).

The basic idea is: Let $\mathbf{V}^k$ be a subspace of $\mathbb{R}^n$ with an orthonormal basis $w_1^k, \ldots, w_m^k$ and W the matrix with columns w_j^k, $S := W^H A W$, $\bar{\lambda}_j^k$ the eigenvalues of S, and T a matrix with the eigenvectors of S as columns. The columns x_j^k of WT are approximations to eigenvectors of A with Ritz values $\bar{\lambda}_j^k = (x_j^k)^H A x_j^k$ that approximate eigenvalues of A. Let us assume that $\bar{\lambda}_{j_s}^k, \ldots, \bar{\lambda}_{j_{s+l-1}}^k \in [\lambda_{\text{lower}}, \lambda_{\text{upper}}]$. For $j \in j_s, \ldots, j_{s+l-1}$ define

$$q_j^k = (A - \bar{\lambda}_j^k I)\, x_j^k\,, \qquad r_j^k = (\bar{A} - \bar{\lambda}_j^k I)^{-1} q_j^k\,, \tag{1}$$

and $\mathbf{V}^{k+1} = \text{span}(\mathbf{V}^k \cup r_{j_s}^k \cup \ldots \cup r_{j_{s+l-1}}^k)$ where $\bar{A}$ is an easy to invert approximation to A ($\bar{A} = \text{diag}(A)$ in [8]). Then $\mathbf{V}^{k+1}$ is an $(m+l)$-dimensional subspace of $\mathbb{R}^n$, and the repetition of the procedure above gives in general improved approximations to eigenvalues and -vectors. Restarting may increase efficiency.

For good convergence, $\mathbf{V}^k$ has to contain crude approximations to all eigenvectors of A with eigenvalues smaller than λ_{lower} [8]. The approximate inverse must not be too accurate, otherwise the method stalls. The reason for this was investigated in [9] and leads to the Jacobi-Davidson (JD) method with an improved definition of r_j^k:

$$[(I - x_j^k (x_j^k)^H)\, (\bar{A} - \bar{\lambda}_j^k I)\, (I - x_j^k (x_j^k)^H)]\, r_j^k = q_j^k\,. \tag{2}$$

The projection $(I - x_j^k (x_j^k)^H)$ in (2) is not easy to incorporate into the matrix, but there is no need to do so, and solving (2) is only slightly more expensive than solving (1).

The method converges quadratically for $\bar{A} = A$.

3 Preconditioning

The character of the JD method is determined by the approximation $\bar{A}$ to A. For obtaining an approximate solution of the preconditioning system (2), we may try an iterative approach [3, 4, 9]. Here, a real symmetric or a complex Hermitian version of the QMR algorithm are used [2, 5, 7] that are directly applied

to the projected system (2) with $\bar{A} = A$. The control of the QMR iteration is as follows. Iteration is stopped when the current residual norm is smaller than the residual norm of QMR in the previous inner JD iteration. By controlling the QMR residual norms, we achieve that the preconditioning system (2) is solved in low accuracy in the beginning and in increasing accuracy in the course of the JD iteration. For a block version of JD, the residual norms of each preconditioning system (2) are separately controlled for each eigenvector to approximate since some eigenvector approximations are more difficult to obtain than others. This adapts the control to the properties of the matrix's spectrum.

Algorithm 1 shows the QMR iteration used to precondition JD for complex Hermitian matrices. The method is derived from the QMR variant described in [5]. Within JD, the matrix B in Algorithm 1 corresponds to the matrix $[(I - x_j^k (x_j^k)^H)\ (A - \bar{\lambda}_j^k I)\ (I - x_j^k (x_j^k)^H)]$ of the preconditioning system (2).

Per QMR iteration, two matrix-vector operations with B and B^* (marked by frames in Algorithm 1) are performed since QMR bases on the non-Hermitian Lanczos algorithm that requires operations with B and $B^T = B^*$ but not with B^H [7]. For real symmetric problems, only one matrix-vector operation per QMR iteration is necessary since then $q^i = Bp^i$ and thus $v^{i+1} = q^i - (\tau^i/\gamma^i)v^i$ hold. The only matrix-vector multiplication to compute per iteration is then Bw^{i+1}.

Naturally, B is not computed element-wise from $[(I - x_j^k (x_j^k)^H)\ (A - \bar{\lambda}_j^k I)\ (I - x_j^k (x_j^k)^H)]$; the operation Bp^i, e.g., is splitted into vector-vector operations and one matrix-vector operation with A.

Note that the framed matrix-vector operations in the complex Hermitian QMR iteration are independent from each other. This can be exploited for a parallel implementation (see 5.2). Moreover, all vector reductions in Algorithm 1 (marked by bullets) are grouped. This in addition makes the QMR variant well suited for a parallel implementation (see 5.3).

4 Storage scheme

Efficient storage schemes for large sparse matrices depend on the sparsity pattern of the matrix, the considered algorithm, and the architecture of the computer system used [1]. Here, the CRS format (Compressed Row Storage) is applied. This format is often used in FE programs and is suited for matrices with regular as well as irregular structure. The principle of the scheme is illustrated in Fig. 1 for a matrix A with non-zeros $a_{i,j}$.

$$A = \begin{pmatrix} a_{1,1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & a_{2,2} & a_{2,3} & 0 & 0 & 0 & 0 & 0 \\ 0 & a_{3,2} & a_{3,3} & a_{3,4} & 0 & 0 & 0 & 0 \\ 0 & 0 & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} & a_{4,8} \\ 0 & 0 & 0 & a_{5,4} & a_{5,5} & 0 & a_{5,7} & 0 \\ 0 & 0 & 0 & a_{6,4} & 0 & a_{6,6} & a_{6,7} & 0 \\ 0 & 0 & 0 & a_{7,4} & a_{7,5} & a_{7,6} & a_{7,7} & 0 \\ 0 & 0 & 0 & a_{8,4} & 0 & 0 & 0 & a_{8,8} \end{pmatrix}$$

Algorithm 1. Complex Hermitian QMR

$p^0 = q^0 = d^0 = s^0 = 0,\ \nu^1 = 1,\ \kappa^0 = -1,\ w^1 = v^1 = r^0 = b - Bx^0$

$\gamma^1 = \|v^1\|,\ \xi^1 = \gamma^1,\ \rho^1 = (w^1)^T v^1,\ \epsilon^1 = (B^* w^1)^T v^1,\ \mu^1 = 0, \tau^1 = \frac{\epsilon^1}{\rho^1}$

$i = 1, 2, \ldots$

$$p^i = \frac{1}{\gamma^i} v^i - \mu^i p^{i-1}$$

$$q^i = \frac{1}{\xi^i} B^* w^i - \frac{\gamma^i \mu^i}{\xi^i} q^{i-1}$$

$$v^{i+1} = \boxed{Bp^i} - \frac{\tau^i}{\gamma^i} v^i$$

$$w^{i+1} = q^i - \frac{\tau^i}{\xi^i} w^i$$

- *if* ($\|r^{i-1}\| <$ *tolerance*) *then STOP*
- $\gamma^{i+1} = \|v^{i+1}\|$
- $\xi^{i+1} = \|w^{i+1}\|$
- $\rho^{i+1} = (w^{i+1})^T v^{i+1}$
- $\epsilon^{i+1} = (\ \boxed{B^* w^{i+1}}\ )^T v^{i+1}$

$$\mu^{i+1} = \frac{\gamma^i \xi^i \rho^{i+1}}{\gamma^{i+1} \tau^i \rho^i}$$

$$\tau^{i+1} = \frac{\epsilon^{i+1}}{\rho^{i+1}} - \gamma^{i+1} \mu^{i+1}$$

$$\theta^i = \frac{|\tau^i|^2 (1 - \nu^i)}{\nu^i |\tau^i|^2 + |\gamma^{i+1}|^2}$$

$$\kappa^i = \frac{-\gamma^i (\tau^i)^* \kappa^{i-1}}{\nu^i |\tau^i|^2 + |\gamma^{i+1}|^2}$$

$$\nu^{i+1} = \frac{\nu^i |\tau^i|^2}{\nu^i |\tau^i|^2 + |\gamma^{i+1}|^2}$$

$$d^i = \theta^i d^{i-1} + \kappa^i p^i$$

$$s^i = \theta^i s^{i-1} + \kappa^i B p^i$$

$$x^i = x^{i-1} + d^i$$

$$r^i = r^{i-1} - s^i$$

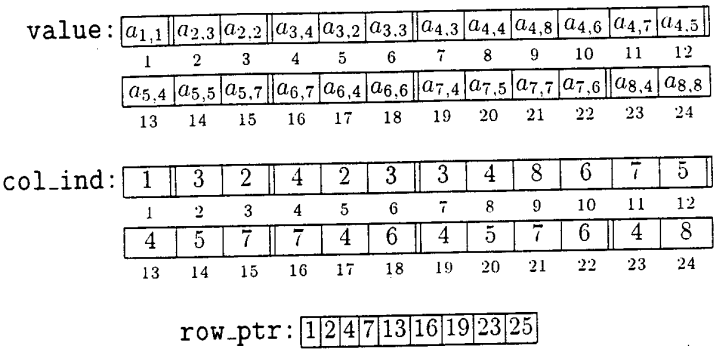


Fig. 1. CRS storage scheme

The non-zeros of matrix A are stored row-wise in three one-dimensional arrays. `value` contains the values of the non-zeros, `col_ind` the corresponding column indices. The elements of `row_ptr` point to the position of the beginning of each row in `value` and `col_ind`.

5 Parallelization Strategies

5.1 Data Distribution

The data distribution scheme considered here balances both matrix-vector and vector-vector operations for irregularly structured sparse matrices on distributed memory systems (see also [2]). The scheme results in a row-wise distribution of the matrix arrays `value` and `col_ind` (see 4); the rows of each processor succeed one another. The distribution of the vector arrays corresponds component-wise to the row distribution of the matrix arrays. In the following, n_k denotes the number of rows of processor k, $k = 0, \ldots, p-1$; n is the total number. g_k is the index of the first row of processor k, and z_i is the number of non-zeros of row i. For these quantities, the following equations hold: $n = \sum_{k=0}^{p-1} n_k$ and $g_k = 1 + \sum_{i=0}^{k-1} n_i$.

In each iteration of an iterative method like JD or QMR, s sparse matrix-vector multiplications and c vector-vector operations are performed. Scalar operations are neglected here. With the data distribution considered, the load generated by row i is proportional to

$$l_i = z_i \cdot s \cdot \zeta + c.$$

The parameter ζ is hardware dependent since it considers the ratio of the costs for a regular vector-vector operation and an irregular matrix-vector operation. However, different matrix patterns could result in different memory access costs, e.g., different caching behavior. Therefore, the parameter ζ is determined at runtime by timings for a row block of the current matrix within the symmetric or

Hermitian QMR solver used. The measurement is performed once on one processor with a predefined number of QMR iterations before the data are distributed. With approximating ζ at run-time for the current matrix, the slight dependence of ζ on the matrix pattern is considered in addition.

For computational load balance, each processor has to perform the p-th fraction of the total number of operations. Hence, the rows of the matrix and the vector components are distributed according to (3).

$$n_k = \begin{cases} \min\limits_{1 \le t \le n-g_k+1} \left\{ t \,\middle|\, \sum\limits_{i=1}^{t} l_{i+g_k-1} \ge \frac{1}{p} \sum\limits_{i=1}^{n} l_i \right\} & \text{for } k = 0, 1, \ldots, q \\ n - \sum\limits_{i=0}^{q} n_i & \text{for } k = q+1 \\ 0 & \text{for } k = q+2, \ldots, p-1 \end{cases} \tag{3}$$

For large sparse matrices and $p \ll n$, usually $q = p - 1$ or $q + 1 = p - 1$ hold. It should be noted that for $\zeta \to 0$ each processor gets nearly the same number of rows and for $\zeta \to \infty$ nearly the same number of non-zeros.

Fig. 2 illustrates the distribution of `col_ind` from Fig. 1 as well as the distribution of the vectors x and y of the matrix-vector multiplication $y = Ax$ to four processors for $\zeta = 5$, $s = 2$, and $c = 13$.

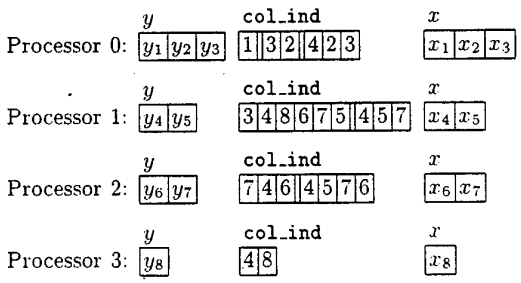


Fig. 2. Data distribution for $\zeta = 5$, $s = 2$, and $c = 13$

In case of an heterogeneous computing environment, e.g., workstation clusters with fast network connections or high-speed connected parallel computers, the data distribution criterion (3) can easily be adapted to different per processor performance or memory resources by predefining weights ω_k per processor k. Only the fraction $1/p$ in (3) has then to be replaced by $\omega_k / \sum_{i=0}^{p-1} \omega_i$.

5.2 Communication Scheme

On a distributed memory system, the computation of the matrix-vector multiplications requires communication because each processor owns only a partial

vector. For the efficient computation of the matrix-vector multiplications, it is necessary to develop a suitable communication scheme (see also [2]). The goal of the scheme is to enable the overlapped execution of computations and data transfers to reduce waiting times based on a parallel matrix pattern analysis and, subsequently, a block rearranging of the matrix data.

First, the arrays `col_ind` (see 4 and 5.1) are analyzed on each processor k to determine which elements result in access to non-local data. Then, the processors exchange information to decide which local data must be sent to which processors. If the matrix-vector multiplications are performed row-wise, components of the vector x of $y = Ax$ are communicated. After the analysis, `col_ind` and `value` are rearranged in such a way that the data that results in access to processor h is collected in block h. The elements of block h succeed one another row-wise with increasing column index per row. Block k is the first block in the arrays `col_ind` and `value` of processor k. Its elements result in access to local data; therefore, in the following, it is called the local block. The goal of this rearranging is to perform computation and communication overlapped. Fig. 3 shows the rearranging for the array `col_ind` of processor 1 from Fig. 2.

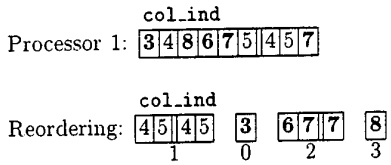


Fig. 3. Rearranging into blocks

The elements of block 1, the local block, result in access to the local components 4 and 5 of x during the row-wise matrix-vector multiplication, whereas operations with the elements of the blocks 0, 2, and 3 require communication with the processors 0, 2, and 3, respectively. For parallel matrix-vector multiplications, each processor first executes asynchronous receive-routines to receive necessary non-local data. Then all components of x that are needed on other processors are sent asynchronously. While the required data is on the network, each processor k performs operations with block k. After that, as soon as non-local data from processor h arrives, processor k continues the matrix-vector multiplication by accessing the elements of block h. This is repeated until the matrix-vector multiplication is complete. Computation and communication are performed overlapped so that waiting times are reduced.

The block structure of the matrix data and the data structures for communications have been optimized for both the real and the complex case to reduce memory requirements and to save unnecessary operations. In addition, cache exploitation is improved by these structures. All message buffers and the block row pointers of the matrix structure are stored in a modified compressed row format. Thus memory requirements per processor almost proportionally decrease

with increasing processor number even if the number of messages per processor markedly rises due to a very irregular matrix pattern.

A parallel preanalysis phase to determine the sizes of all data structures preceeds the detailed communication analysis and the matrix rearranging. This enables dynamic memory allocation and results in a further reduction of memory requirements since memory not needed any more, e.g., after the analysis phases, can be deallocated. Another advantage is that the same executable can be used for problems of any structure and size.

For complex Hermitian problems, two independent matrix-vector products with B and B^* have to be computed per QMR iteration (see the framed operations in Algorithm 1). Communications for both operations — they possess the same communication scheme — are coupled to reduce communication overhead and waiting times.

The data distribution and the communication scheme presented here do not require any knowledge about a specific discretization mesh; the schemes are determined automatically by the analysis of the indices of the non-zero matrix elements.

5.3 Synchronization

Synchronization overhead is reduced by grouping inner products and norm computations within the QMR and the JD iteration. For QMR in both the real symmetric and the complex Hermitian case, special parallel variants based on [5] have been developed that require only one synchronization point per iteration step. For a parallel message passing implementation of Algorithm 1, all local values of the vector reductions marked by bullets can be included into one global communication to determine the global values.

6 RESULTS

All parts of the algorithms have been investigated with various application problems on the massively parallel systems NEC Cenju-3 with up to 128 processors (64 Mbytes main memory per processor) and Cray T3E with up to 512 processors (128 Mbytes main memory per processor). The codes have been written in FORTRAN 77 and C; MPI is used for message passing.

6.1 Numerical test cases

Numerical and performance tests of the JD implementation have been carried out with the large sparse real symmetric matrices **Episym1** to **Episym6** and the large sparse complex Hermitian matrices **Epiherm1** and **Epiherm2** stemming from the simulation of electron/phonon interaction [10], with the real symmetric matrices **Struct1** to **Struct3** from structural mechanics problems (finite element discretization), and with the dense complex Hermitian matrix **Thinfilms** from

the simulation of thin films with defects. The smaller real symmetric test matrices **Laplace**, **GregCar**, **CulWil**, and **RNet** originate from finite difference discretization problems. Table 1 gives a survey of all matrices considered.

Table 1. Numerical data of the considered large sparse matrices

Matrix	Properties	Order	Number of non-zeros
Episym1	Real symmetric	98,800	966,254
Episym2	Real symmetric	126,126	1,823,812
Episym3	Real symmetric	342,200	3,394,614
Episym4	Real symmetric	1,009,008	14,770,746
Episym5	Real symmetric	5,513,508	81,477,386
Episym6	Real symmetric	11,639,628	172,688,506
Epiherm1	Complex Hermitian	126,126	1,823,812
Epiherm2	Complex Hermitian	1,009,008	14,770,746
Thinfilms	Complex Hermitian	1,413	1,996,569
Struct1	Real symmetric	835	13,317
Struct2	Real symmetric	2,839	299,991
Struct3	Real symmetric	25,222	3,856,386
Laplace	Real symmetric	900	7,744
GregCar	Real symmetric	1,000	2,998
CulWil	Real symmetric	1,000	3,996
RNet	Real symmetric	1,000	6,400

6.2 Effect of Preconditioning

For the following investigation about the effect on QMR preconditioning on JD, the JD iteration was stopped if the residual norms divided by the initial norms are less than 10^{-5}.

In Fig. 4, times for computing the four smallest eigenvalues and -vectors of the two real symmetric matrices **Episym2** and **Struct3** on 64 NEC Cenju-3 processors are compared for different preconditioners.

The best results are gained for JD with adaptive QMR preconditioning and a few preceding, diagonally preconditioned outer JD steps (4 or 1). Compared with pure diagonal preconditioning, the number of matrix-vector multiplications required decreases from 6,683 to 953 for the matrix **Episym2** from electron/phonon interaction. Note that the Lanczos algorithm used in the application code requires about double the number of matrix-vector multiplications as QMR preconditioned JD for this problem. For the matrix **Struct3** from structural mechanics, the diagonally preconditioned method did not converge in

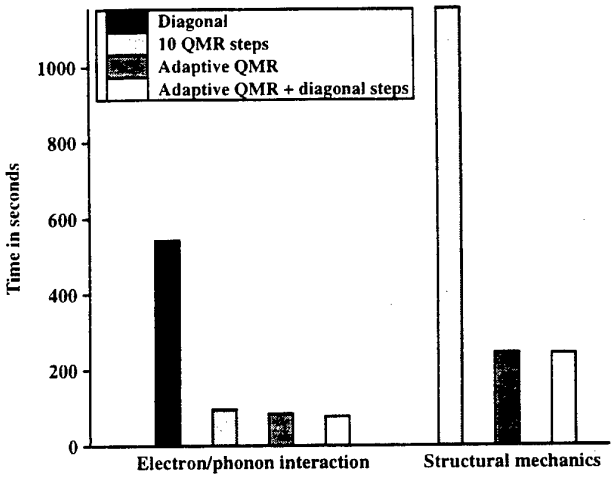


Fig. 4. Different preconditioners. Real symmetric matrices **Episym2** and **Struct3**. NEC Cenju-3, 64 processors

100 minutes. Note that in this case the adaptive approach is markedly superior to preconditioning with a fixed number of 10 QMR iterations; the number of matrix-vector multiplications decreases from 55,422 to 11,743.

6.3 JD versus Lanczos Method

In Table 2, the sequential execution times on an SGI O^2 workstation (128 MHz, 128 Mbytes main memory) of a common implementation of the symmetric Lanczos algorithm [6] and adaptively QMR preconditioned JD are compared for computing the four smallest eigenvalues and -vectors. In both cases, the matrices are stored in CRS format. The required accuracy of the results was set close to machine precision. Except for matrix **CulWil** — the Lanczos algorithm did not converge within 120 minutes since the smallest eigenvalues of the matrix are very close to each other — both methods gave the same eigenvalues and -vectors within the required accuracy. Since the Lanczos method applied stores all Lanczos vectors to compute the eigenvectors of A only the smallest matrices from Table 1 could be used for the comparison.

Table 2 shows that the JD method is markedly superior to the Lanczos algorithm for the problems tested. Moreover, the results for the matrices **Laplace** and **CulWil**— some of the smallest eigenvalues are very close to each other — appear to indicate that JD can handle the problem of close eigenvalues much better than the Lanczos algorithm.

Table 2. Comparison of JD and the symmetric Lanczos algorithm. Sequential execution times. SGI O^2 workstation

Matrix	Lanczos	JD	Ratio
Struct1	79.3 s	34.4 s	2.3
Struct2	5633.4 s	899.7 s	6.3
Laplace	97.4 s	2.7 s	36.1
GregCar	95.9 s	15.7 s	6.1
CulWil	—	16.1 s	—
RNet	197.9 s	41.8 s	4.7

6.4 Parallel Performance

In all following investigations, the JD iteration was stopped if the residual norms divided by the initial norms are less than 10^{-5} (10^{-10} for the Cray T3E results).

Fig. 5 shows the scaling of QMR preconditioned JD for computing the four smallest eigenpairs of the large real symmetric electron/phonon interaction matrices **Episym1**, **Episym3**, and **Episym4** on NEC Cenju-3. On 128 processors, speedups of 45.5, 105.7, and 122.3 are achieved for the matrices with increasing order; the corresponding execution times are 8.3 s, 23.3 s, and 168.3 s.

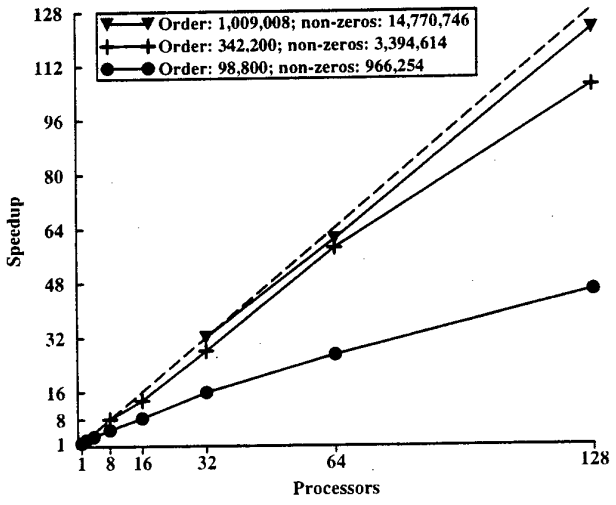


Fig. 5. Speedups. Real symmetric matrices **Episym1**, **Episym3**, and **Episym4**, electron/phonon interaction. NEC Cenju-3

In Fig. 6, speedups of QMR preconditioned JD for computing the four smallest eigenpairs of the two large real symmetric electron/phonon interaction matri-

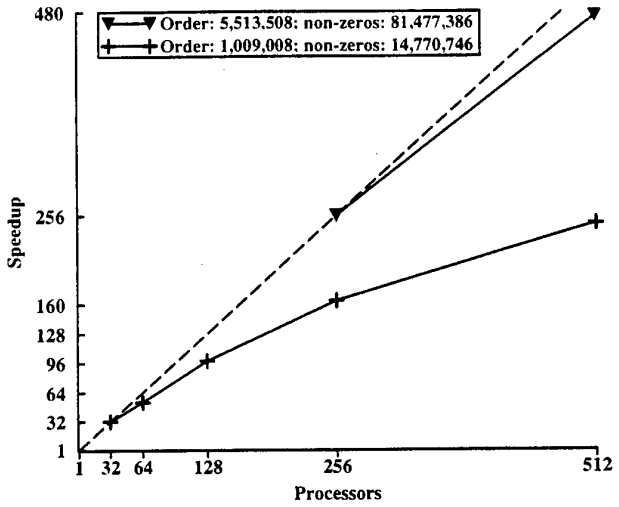


Fig. 6. Speedups. Real symmetric matrices **Episym4** and **Episym5**, electron/phonon interaction. Cray T3E

ces **Episym4** and **Episym5** on Cray T3E are displayed. The problems **Episym4** and **Episym5** of order 1,009,008 and 5,513,508 result in execution times of 15.2 s and 129.2 s, respectively, on 512 processors. The largest real symmetric electron/phonon interaction problem **Episym6** computed of order 11,639,628 has an execution time of 278.7 s on 512 processors.

The effect of coupling the communication for the two independent matrix-vector multiplications per complex Hermitian QMR iteration (see the framed operations in Algorithm 1) is displayed in Fig. 7 for computing the four smallest eigenpairs of the dense complex Hermitian matrix **Thinfilms**. This matrix is chosen since the problem is of medium size and the matrix-vector operations require communication with all non-local processors. In Fig. 7, the execution times on NEC Cenju-3 of JD with and without coupling divided by the total number of matrix-vector products (MVPs) are compared.

Communication coupling halves the number of messages and doubles the message length. By this, the overhead of communication latencies is markedly reduced, and possibly a higher transfer rate can be reached. For the matrix **Thinfilms**, coupling gives a gain of 5% to 15% of the total time. For much larger matrices, gains are usually very slight since if the message lengths are big latency is almost negligible and higher tranfer rates cannot be reached. For the matrix **Episym2**, e.g., corresponding timings on 128 processors give 64.5 ms without coupling and 64.4 ms with coupling.

Fig. 8 shows the scaling of the complex Hermitian version of QMR preconditioned JD for computing the four smallest eigenpairs of the two large complex Hermitian electron/phonon interaction matrices **Epiherm1** and **Epiherm2** on

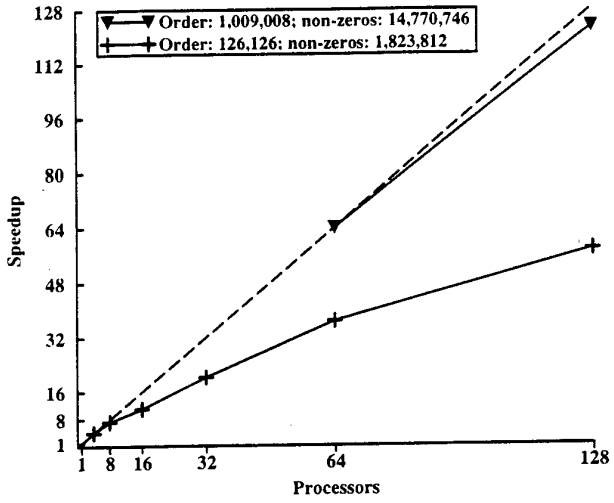


Fig. 8. Speedups. Complex Hermitian matrices **Epiherm1** and **Epiherm2**. electron/phonon interaction. NEC Cenju-3

Systems: Building Blocks for Iterative Methods. SIAM. Philadelphia (1993)
2. Basermann, A.: QMR and TFQMR Methods for Sparse Nonsymmetric Problems on Massively Parallel Systems. In: Renegar, J., Shub, M., Smale, S. (eds.): The Mathematics of Numerical Analysis, series: Lectures in Applied Mathematics, Vol. 32. AMS (1996) 59–76
3. Basermann, A., Steffen, B.: New Preconditioned Solvers for Large Sparse Eigenvalue Problems on Massively Parallel Computers. In: Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing (CD-ROM). SIAM, Philadelphia (1997)
4. Basermann, A., Steffen, B.: Preconditioned Solvers for Large Eigenvalue Problems on Massively Parallel Computers and Workstation Clusters. Technical Report FZJ-ZAM-IB-9713. Research Centre Jülich GmbH (1997)
5. Bücker, H.M., Sauren, M.: A Parallel Version of the Quasi-Minimal Residual Method Based on Coupled Two-Term Recurrences. In: Lecture Notes in Computer Science, Vol. 1184. Springer (1996) 157–165
6. Cullum, J.K., Willoughby, R.A.: Lanczos Algorithms for Large Symmetric Eigenvalue Computations, Volume I: Theory. Birkhäuser, Boston Basel Stuttgart (1985)
7. Freund, R.W., Nachtigal, N.M.: QMR: A Quasi-Minimal Residual Method for Non-Hermitian Linear Systems. Numer. Math. **60** (1991) 315–339
8. Kosugi, N.: Modifications of the Liu-Davidson Method for Obtaining One or Simultaneously Several Eigensolutions of a Large Real Symmetric Matrix. Comput. Phys. **55** (1984) 426–436
9. Sleijpen, G.L.G., van der Vorst, H.A.: A Jacobi-Davidson Iteration Method for Linear Eigenvalue Problems. SIAM J. Matrix Anal. Appl. **17** (1996) 401–425
10. Wellein, G., Röder, H., Fehske, H.: Polarons and Bipolarons in Strongly Interacting Electron-Phonon Systems. Phys. Rev. B **53** (1996) 9666–9675

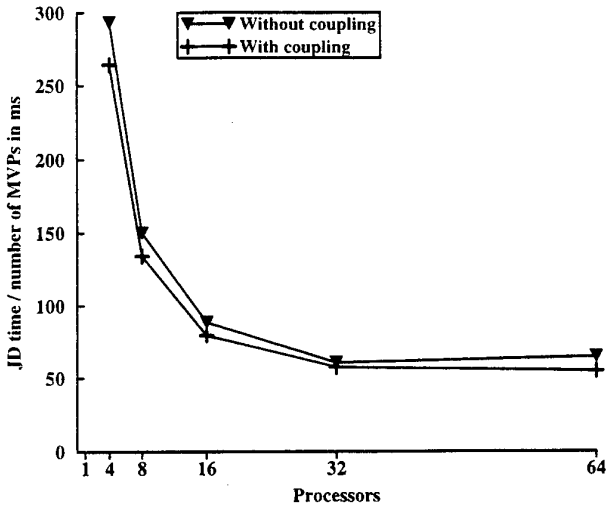


Fig. 7. Communication coupling. Complex Hermitian matrix **Thinfilms**. NEC Cenju-3

NEC Cenju-3. On 128 processors, speedups of 57.5 (execution time 38.8 s) and 122.5 (execution time 320.2 s) are achieved for the matrices **Epiherm1** and **Epiherm2** of order 126,126 and 1,009,008, respectively.

7 CONCLUSIONS

By real symmetric and complex Hermitian matrices from applications, the efficiency of the developed parallel JD methods was demonstrated on massively parallel systems. The data distribution strategy applied supports computational load balance for both irregular matrix-vector and regular vector-vector operations in iterative solvers. The investigated communication scheme for matrix-vector multiplications together with a block rearranging of the sparse matrix data makes possible the overlapped execution of computations and data transfers. Moreover, parallel adaptive iterative preconditioning with QMR was shown to accelerate JD convergence markedly. Coupling the communications for the two independent matrix-vector products in the complex Hermitian QMR iteration halves the number of required messages and results in additional execution time gains for small and medium size problems. Furthermore, a sequential comparison of QMR preconditioned JD and the symmetric Lanczos algorithm indicates a superior convergence and time behavior in favor of JD.

References

1. Barrett, R., Berry, M., Chan, T., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., van der Vorst, H.: Templates for the Solution of Linear

Comparisons of Parallel Algorithms to Evaluate Orthogonal Series

Roberto Barrio

GME, Departamento de Matemática Aplicada, CPS,
Univ. Zaragoza, E–50015 Zaragoza. Spain
rabarrio@posta.unizar.es

Abstract. New parallel algorithms for the evaluation of series of orthogonal polynomials are presented. The performance of these algorithms on a message passing distributed memory computer (a Cray T3D) is compared.

1 Introduction

The evaluation of polynomials is one of the most common problems in scientific computing. Therefore, it has been extensively studied and several algorithms suitable for parallel evaluation have been proposed [6, 9–11]. All these algorithms focus their attention on the evaluation of power series.

In several scientific applications polynomials do not appear as power series, but are written using orthogonal polynomials [1] due to their special features. A parallel algorithm was presented in [2, 3] for the evaluation of Chebyshev series.

In this paper we present two new algorithms for the evaluation of polynomials written as finite series of orthogonal polynomials. These algorithms are based on the matrix formulation of the sequential algorithms and afterwards they apply some techniques used in the parallel solution of tridiagonal [14] and banded linear systems [5, 8, 12]. These algorithms are given in Section 3 and compared in Section 4 on a Cray T3D.

A sequence of orthogonal polynomials $\{\phi_r(x)\}$ always satisfies [1] the triple recurrence relation:

$$\phi_r(x) - \alpha_r(x)\,\phi_{r-1}(x) - \beta_r\,\phi_{r-2}(x) = 0, \qquad r \geq 2, \tag{1}$$

for some functions $\alpha_r(x)$ and β_r. Sequential algorithms for evaluation of finite series based on orthogonal polynomials exist and are extensively used, such as the Clenshaw [4] or Forsythe [7] algorithms.

The Forsythe algorithm [7] is based on a direct application of the three–term recurrence formula (1) and consists of:

$$\sum_{r=0}^{n} c_r \phi_r(x) = f_n(x), \tag{2}$$

where

$$\left.\begin{aligned} &\phi_0(x), \quad \phi_1(x), \quad f_1(x) = c_0\,\phi_0(x) + c_1\,\phi_1(x), \\ &\phi_r(x) = \alpha_r(x)\,\phi_{r-1}(x) + \beta_r\,\phi_{r-2}(x), \\ &f_r(x) = f_{r-1}(x) + c_r\,\phi_r(x), \end{aligned}\right\} \quad r = 2\ldots,n. \tag{3}$$

A further computational algorithm is the Clenshaw algorithm [4, 13], that permits evaluation of a finite series of orthogonal polynomials by means of the expression:

$$\sum_{r=0}^{n} c_r\phi_r(x) = \{c_0 + \beta_2\,q_2(x)\}\;\phi_0(x) + q_1(x)\,\phi_1(x) \tag{4}$$

where

$$\begin{aligned} &q_{n+1}(x) = q_{n+2}(x) = 0, \\ &q_r(x) = c_r + \alpha_{r+1}(x)\,q_{r+1}(x) + \beta_{r+2}\,q_{r+2}(x), \qquad \text{for} \quad r = n,\ldots,1. \end{aligned} \tag{5}$$

2 Parallel Algorithm to Evaluate Finite Series of Chebyshev Polynomials

The parallel algorithms to evaluate Chebyshev series in [2, 3] are based on the product rules for the first ($T_i(x)$) and the second kind ($U_i(x)$) Chebyshev polynomials:

$$\left.\begin{aligned} T_{m+p}(x) &= 2\,T_p(x)\,T_m(x) - T_{m-p}(x), \\ U_{m+p}(x) &= 2\,T_p(x)\,U_m(x) - U_{m-p}(x), \end{aligned}\right\} \text{for} \quad m \geq p. \tag{6}$$

The parallel Forsythe algorithm to evaluate $p_n^I(x) = \sum_{r=0}^{n} c_r\,T_r(x)$ or $p_n^{II}(x) = \sum_{r=0}^{n} c_r\,U_r(x)$ can be written, with $n = k\,p - 1$, as the following routine [2]:

STEP I: **Processor** m $(m = 0,\ldots,p-1)$
- $t_0 = 1$
- **if** $(p_n^I(x))$ **then**
  - $t_1 = x$
- **else if** $(p_n^{II}(x))$ **then**
  - $t_1 = 2\,x$
- **end if**
- **for** $i = 2, 2p-1$
  - $t_i = 2\,x\,t_{i-1} - t_{i-2}$
- **end**

STEP II: **Processor** m $(m = 0,\ldots,p-1)$
- $f_m = c_m\,t_m + c_{p+m}\,t_{m+p}$
- **for** $i = 2, k$
  - $t_{ip+m} = 2\,t_p\,t_{(i-1)p+m} - t_{(i-2)p+m}$
  - $f_m = f_m + c_{ip+m}\,t_{ip+m}$
- **end**

STEP III: **Processor** m $(m = 0, \ldots, p-1)$
red_sum_0(f_m, sum)

In the algorithm, the number p is the number of processors and the function **red_sum_0**(f_m, sum) stands for a global reduction operation, in this case the addition of f_m for $m = 0, \ldots, p-1$, and writes the result in the variable sum at the processor 0. The variable sum gives as output the value of the polynomial.

3 Parallel Algorithms to Evaluate Finite Series of General Orthogonal Polynomials

The algorithm to evaluate Chebyshev series is based on the product rules for the Chebyshev polynomials. These rules are very simple in this special case and permit us to obtain an efficient parallel algorithm. Unfortunately, other kinds of orthogonal polynomials satisfy very cumbersome product rules. Therefore, in these cases we must obtain parallel algorithms in a different way.

First of all, we may remark that for general families of orthogonal polynomials the coefficients in the recurrence relation (1) may be calculated at the same time we evaluate the finite series. This process may be done in parallel.

In Table 1 we show (Abramowitz *et al.*, [1]) the coefficients $\alpha_r(x)$ and β_r in the case of the Jacobi polynomials $P_i^{(\alpha,\beta)}(x)$, Gegenbauer polynomials $C^{\lambda}(x)_i$, Legendre polynomials $P_i(x)$ and Chebyshev polynomials of the first $T_i(x)$ and second kind $U_i(x)$.

Table 1. Coefficients of the triple recurrence relation for some families of orthogonal polynomials.

	$\alpha_r(x)$	β_r
$P_i^{(\alpha,\beta)}(x)$	$x\,\frac{(2r+\alpha+\beta)(2r+\alpha+\beta-1)}{2\,r\,(r+\alpha+\beta)} + \frac{(\alpha^2-\beta^2)(2r+\alpha+\beta-1)}{2\,r\,(r+\alpha+\beta)(2r+\alpha+\beta-2)}$	$-\frac{(r+\alpha-1)(r+\beta-1)(2r+\alpha+\beta)}{r\,(r+\alpha+\beta)(2r+\alpha+\beta-2)}$
$C_i^{\lambda}(x)$	$2\,x\,\frac{r-1+\lambda}{r}$	$-\frac{r-2+2\,\lambda}{r}$
$P_i(x)$	$x\,\frac{2\,r-1}{r}$	$-\frac{r-1}{r}$
$T_i(x)$	$2\,x$	-1
$U_i(x)$	$2\,x$	-1

3.1 Parallel Clenshaw's Algorithm

The Clenshaw algorithm (Eqs. (4),(5)) can be formulated using matrix notation. Let C be the matrix

$$C = \begin{pmatrix} 1 & -\alpha_2 & -\beta_3 & & & \\ & 1 & -\alpha_3 & \ddots & & \\ & & \ddots & \ddots & -\beta_n & \\ & & & \ddots & -\alpha_n & \\ & & & & 1 & \end{pmatrix}, \tag{7}$$

then the Clenshaw algorithm is equivalent to solve the banded upper triangular linear system $C\,q = c$ where q and c are the vectors $q^T = (q_1, q_2, \ldots, q_n)$ and $c^T = (c_1, c_2, \ldots, c_n)$ and afterwards to use the relation (4) to obtain the value of the series.

To simplify the notation we suppose that $n = k\,p$, being p the number of processors. To illustrate the algorithm we present the case $n = 12$ and $p = 3$, then the matrix C (7), written as a block matrix, will be

$$C = \left(\begin{array}{cccc|cccc|cccc} 1 & -\alpha_2 & -\beta_3 & & & & & & & & & \\ & 1 & -\alpha_3 & -\beta_4 & & & & & & & & \\ & & 1 & -\alpha_4 & -\beta_5 & & & & & & & \\ & & & 1 & -\alpha_5 & -\beta_6 & & & & & & \\ \hline & & & & 1 & -\alpha_6 & -\beta_7 & & & & & \\ & & & & & 1 & -\alpha_7 & -\beta_8 & & & & \\ & & & & & & 1 & -\alpha_8 & -\beta_9 & & & \\ & & & & & & & 1 & -\alpha_9 & -\beta_{10} & & \\ \hline & & & & & & & & 1 & -\alpha_{10} & -\beta_{11} & \\ & & & & & & & & & 1 & -\alpha_{11} & -\beta_{12} \\ & & & & & & & & & & 1 & -\alpha_{12} \\ & & & & & & & & & & & 1 \end{array}\right).$$

Now, we may use in parallel the Gaussian elimination to diagonalize each of the diagonal submatrices, that is, we apply the *divide and conquer* algorithm (Wang [14]), and we obtain the system $C^R \cdot q = c^R$ where

$$C^R = \left(\begin{array}{cccc|cccc|cccc} 1 & & & & a_0^3 & b_0^3 & & & & & & \\ & 1 & & & a_0^2 & b_0^2 & & & & & & \\ & & 1 & & a_0^1 & b_0^1 & & & & & & \\ & & & 1 & a_0^0 & b_0^0 & & & & & & \\ \hline & & & & 1 & & & & a_1^3 & b_1^3 & & \\ & & & & & 1 & & & a_1^2 & b_1^2 & & \\ & & & & & & 1 & & a_1^1 & b_1^1 & & \\ & & & & & & & 1 & a_1^0 & b_1^0 & & \\ \hline & & & & & & & & 1 & & & \\ & & & & & & & & & 1 & & \\ & & & & & & & & & & 1 & \\ & & & & & & & & & & & 1 \end{array}\right), \quad q = \begin{pmatrix} q_1 \\ q_2 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ q_{12} \end{pmatrix}, \quad c^R = \begin{pmatrix} c_1^R \\ c_2^R \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_{12}^R \end{pmatrix} \tag{8}$$

Thus, our problem is to solve a reduced linear system of order $2\,p$ (in this case 6)

$$\left(\begin{array}{cc|cc|cc} 1 & & a_0^3 & b_0^3 & & \\ & 1 & a_0^2 & b_0^2 & & \\ \hline & & 1 & & a_1^3 & b_1^3 \\ & & & 1 & a_1^2 & b_1^2 \\ \hline & & & & 1 & \\ & & & & & 1 \end{array}\right) \left(\begin{array}{c} q_1 \\ q_2 \\ q_5 \\ q_6 \\ q_9 \\ q_{10} \end{array}\right) = \left(\begin{array}{c} c_1^R \\ c_2^R \\ c_5^R \\ c_6^R \\ c_9^R \\ c_{10}^R \end{array}\right) \tag{9}$$

And finally, with the values of q_1 and q_2, we can evaluate the polynomial using (4).

Therefore, the complete algorithm consists of:

STEP I: **Processor** m $(m = 0, \ldots, p-1)$
  for $i = m\,k, (m+1)\,k - 1$
    evaluate$(\alpha_{i+2}, \beta_{i+3})$
  end

STEP II: **Processor** $p - 1$

$c^1 = c_n$
$c^2 = c_{n-1} + \alpha_n\, c^1$
$c^3 = c_{n-2} + \beta_n\, c^1$
for $i = 1, k-3$
  $c^1 = c^2$
  $c^2 = c^3 + \alpha_{n-i}\, c^1$
  $c^3 = c_{n-i-2} + \beta_{n-i}\, c^1$
end
$c^R_{(p-1)k+2} = c^2$
$c^R_{(p-1)k+1} = c^3 + \alpha_{n-k+2}\, c^2$
$a^{k-2}_{p-1} = 0$
$a^{k-1}_{p-1} = 0$
$b^{k-2}_{p-1} = 0$
$b^{k-1}_{p-1} = 0$

Processor $m \neq p - 1$

$a^1 = -\alpha_{(m+1)k+1}$
$a^2 = -\beta_{(m+1)k+1} + \alpha_{(m+1)k}\, a^1$
$a^3 = \beta_{(m+1)k}\, a^1$
$b^1 = -\beta_{(m+1)k+2}$
$b^2 = \alpha_{(m+1)k}\, b^1$
$b^3 = \beta_{(m+1)k}\, b^1$
$c^1 = c_{(m+1)k}$
$c^2 = c_{(m+1)k-1} + \alpha_{(m+1)k}\, c^1$
$c^3 = c_{(m+1)k-2} + \beta_{(m+1)k}\, c^1$
for $i = 1.\, k-3$
  $a^1 = a^2$

$a^2 = a^3 + \alpha_{(m+1)k-i}\, a^1$
$a^3 = \beta_{(m+1)k-i}\, a^1$
$b^1 = b^2$
$b^2 = b^3 + \alpha_{(m+1)k-i}\, b^1$
$b^3 = \beta_{(m+1)k-i}\, b^1$
$c^1 = c^2$
$c^2 = c^3 + \alpha_{(m+1)k-i}\, c^1$
$c^3 = c_{(m+1)k-i-2} + \beta_{(m+1)k-i}\, c^1$
end
$a_m^{k-2} = a^2$
$a_m^{k-1} = a^3 + \alpha_{mk+2}\, a^2$
$b_m^{k-2} = b^2$
$b_m^{k-1} = b^3 + \alpha_{mk+2}\, a^2$
$c_{mk+2}^R = c^2$
$c_{mk+1}^R = c^3 + \alpha_{mk+2}\, c^2$

STEP III: **Processor** m $(m = 0, \ldots, p-1)$
communicate_0$(a_m^{k-2}, a_m^{k-1}, b_m^{k-2}, b_m^{k-1}, c_{mk+2}^R, c_{mk+1}^R)$

STEP IV: **Processor** 0
$q_3 = c_{(p-1)k+1}^R$
$q_4 = c_{(p-1)k+2}^R$
for $m = p-2, 0$ **step** -1
$q_1 = c_{m\,k+1}^R - a_m^{k-1}\, q_3 - b_m^{k-1}\, q_4$
$q_2 = c_{m\,k+2}^R - a_m^{k-2}\, q_3 - b_m^{k-2}\, q_4$
$q_3 = q_1$
$q_4 = q_2$
end

STEP V: **Processor** 0
$sum = (c_0 + \beta_2\, q_2)\, \phi_0(x) + q_1\, \phi_1(x)$

Where the function **evaluate**(α_i,β_i) evaluates the values of the coefficients (α_i,β_i) and **communicate_i**(var) communicates to processor **i** the variables var.

In this algorithm we only need the value of the polynomial, that is, we only need the terms q_1 and q_2 of the solution of the linear system $C\,q = c$. For this reason we only have one communication process.

In the complexity analysis of the algorithm we suppose that the evaluation of the coefficients α_i and β_i have a computational complexity T_α and T_β, and T_{com} is the complexity of each communication process. Thus, a simple analysis of the algorithm gives us its computational complexity.

Proposition 1. *The theoretical computational complexity of the parallel Clenshaw algorithm is*

$$T_p = \lceil \frac{n}{p} \rceil\, (10 + T_\alpha + T_\beta) + 8\,p - 19 + T_{com}.$$

3.2 Parallel Forsythe's Algorithm

In the Forsythe algorithm (Eqs. (2),(3)), the evaluation of the orthogonal polynomials $\{\phi_i\}$ is equivalent to solving the linear system $F\,\phi = e_{n+1}$, where

$$F = \begin{pmatrix} 1 & -\alpha_n & -\beta_n & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & -\beta_2 & \\ & & & \ddots & -\alpha_1 & \\ & & & & 1 & \end{pmatrix}, \phi = \begin{pmatrix} \phi_n \\ \vdots \\ \vdots \\ \phi_1 \\ \phi_0 \end{pmatrix}, e_{n+1} = \begin{pmatrix} 0 \\ \vdots \\ \vdots \\ 0 \\ \phi_0 \end{pmatrix}, \qquad (10)$$

Afterwards we only need to perform the sum $\sum c_r\,\phi_r$.

To simplify the notation we suppose that $n = k\,p - 1$, where p is the number of processors. To illustrate the algorithm we present the case $n = 11$ and $p = 3$, then the matrix F (10) will be

$$F = \left(\begin{array}{cccc|cccc|cccc} 1 & -\alpha_{11} & -\beta_{11} & & & & & & & & & \\ & 1 & -\alpha_{10} & -\beta_{10} & & & & & & & & \\ & & 1 & -\alpha_9 & -\beta_9 & & & & & & & \\ & & & 1 & -\alpha_8 & -\beta_8 & & & & & & \\ \hline & & & & 1 & -\alpha_7 & -\beta_7 & & & & & \\ & & & & & 1 & -\alpha_6 & -\beta_6 & & & & \\ & & & & & & 1 & -\alpha_5 & -\beta_5 & & & \\ & & & & & & & 1 & -\alpha_4 & -\beta_4 & & \\ \hline & & & & & & & & 1 & -\alpha_3 & -\beta_3 & \\ & & & & & & & & & 1 & -\alpha_2 & -\beta_2 \\ & & & & & & & & & & 1 & -\alpha_1 \\ & & & & & & & & & & & 1 \end{array}\right).$$

As before, by using the Gaussian elimination and the *divide and conquer* algorithm we obtain the system $F^R \cdot \phi = e^R_{n+1}$, where we write F^R using the same notation as C^R (8) and

$$\phi = (\phi_{11}, \phi_{10}, \ldots, \phi_0)^T, \qquad e^R_{n+1} = \left(0, 0, \ldots, 0, c^R_3, c^R_2, c^R_1, c^R_0\right)^T \qquad (11)$$

Our problem is reduced to solving the linear system of order 6,

$$\left(\begin{array}{cc|cc|cc} 1 & & a^3_0 & b^3_0 & & \\ & 1 & a^2_0 & b^2_0 & & \\ \hline & 1 & & & a^3_1 & b^3_1 \\ & & 1 & & a^2_1 & b^2_1 \\ \hline & & & & 1 & \\ & & & & & 1 \end{array}\right) \begin{pmatrix} \phi_{11} \\ \phi_{10} \\ \phi_7 \\ \phi_6 \\ \phi_3 \\ \phi_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ c^R_3 \\ c^R_2 \end{pmatrix}. \qquad (12)$$

Then we communicate the solutions to each processor in order to obtain the values of all the orthogonal polynomials $\{\phi_i(x)\}$ by solving the different subsystems. Finally, we obtain the value of the polynomial by adding each partial sum.

Thus, the complete algorithm consists on:

STEP I: **Processor** m $(m = 0, \ldots, p-1)$
  for $i = (p-m-1)\,k, (p-m)\,k-1$
    evaluate(α_i, β_i)
  end

STEP II: **Processor** $p-1$
  $c_0^R = 1$
  $c_1^R = \alpha_1$
  $c_2^R = \beta_2$
  for $i = 1, k-3$
    $c_{i+1}^R = c_{i+1}^R + \alpha_{i+1}\, c_i^R$
    $c_{i+2}^R = \beta_{i+2}\, c_i^R$
  end
  $c_{k-1}^R = c_{k-1}^R + \alpha_{k-1}\, c_{k-2}^R$
  $a_{p-1}^{k-2} = 0$
  $a_{p-1}^{k-1} = 0$
  $b_{p-1}^{k-2} = 0$
  $b_{p-1}^{k-1} = 0$
 Processor $m \neq p-1$
  $a_m^0 = -\alpha_{(p-m-1)k}$
  $a_m^1 = -\beta_{(p-m-1)k+1} + \alpha_{(p-m-1)k+1}\, a_m^0$
  $a_m^2 = \beta_{(p-m-1)k+2}\, a_m^0$
  $b_m^0 = -\beta_{(p-m-1)k}$
  $b_m^1 = \alpha_{(p-m-1)k+1}\, b_m^0$
  $b_m^2 = \beta_{(p-m-1)k+2}\, b_m^0$
  for $i = 1, k-3$
    $a_m^{i+1} = a_m^{i+1} + \alpha_{(p-m-1)k+i+1}\, a_m^i$
    $a_m^{i+2} = \beta_{(p-m-1)k+i+2}\, a_m^i$
    $b_m^{i+1} = b_m^{i+1} + \alpha_{(p-m-1)k+i+1}\, b_m^i$
    $b_m^{i+2} = \beta_{(p-m-1)k+i+2}\, b_m^i$
  end
  $a_m^{k-1} = a_m^{k-1} + \alpha_{(p-m)k-1}\, a_m^{k-2}$
  $b_m^{k-1} = b_m^{k-1} + \alpha_{(p-m)k-1}\, b_m^{k-2}$
  $c_{(p-m)k-2}^R = 0$
  $c_{(p-m)k-1}^R = 0$

STEP III: **Processor** m $(m = 0, \ldots, p-1)$
  communicate_0$(a_m^{k-2}, a_m^{k-1}, b_m^{k-2}, b_m^{k-1}, c_{(p-m)k-2}^R, c_{(p-m)k-1}^R)$

STEP IV: **Processor** 0
  $\phi_{k-2} = c_{k-2}^R$
  $\phi_{k-1} = c_{k-1}^R$
  for $j = 2, p$
    $\phi_{jk-2} = -a_{p-j}^{k-2}\, \phi_{(j-1)k-1} - b_{p-j}^{k-2}\, \phi_{(j-1)k-2}$
    $\phi_{jk-1} = -a_{p-j}^{k-1}\, \phi_{(j-1)k-1} - b_{p-j}^{k-1}\, \phi_{(j-1)k-2}$
  end

$\phi_{-2} = 0$
$\phi_{-1} = 0$

STEP V: **Processor** 0
communicate_m$(\phi_{(p-m-1)k-2}, \phi_{(p-m-1)k-1})$

STEP VI: **Processor** $p-1$
$s_{p-1} = 0$
for $i = 0, k-1$
$\phi_i = c_i^R$
$s_{p-1} = s_{p-1} + c_i \phi_i$
end
Processor $m \neq p-1$
$s_m = 0$
for $i = 0, k-1$
$\phi_{(p-m-1)k+i} = -\phi_{(p-m-1)k-1}\, a_m^i - \phi_{(p-m-1)k-2}\, b_m^i$
$s_m = s_m + c_{(p-m-1)k+i}\, \phi_{(p-m-1)k+i}$
end

STEP VII: **Processor** m $(m = 0, \ldots, p-1)$
red_sum_0(s_m, sum)

As before, a simple analysis of the algorithm gives us its computational complexity.

Proposition 2. *The theoretical computational complexity of the parallel Forsythe algorithm is*

$$T_p = \lceil \frac{n}{p} \rceil \, (11 + T_\alpha + T_\beta) + 6\,p - 15 + 3\,T_{com}.$$

This algorithm has a greater computational complexity than the parallel Clenshaw and also it requires more global communication processes.

4 Numerical Tests

The algorithms presented here have been tested on a Cray T3D at the Edinburgh Parallel Computing Centre (EPCC), using up to 128–PE with Message Passing Interface (MPI) as the parallel environment. This computer is hosted by a Cray Y–MP system. Each T3D PE consists of a DECchip 21064 Alpha processor with 64Mb of memory.

In Figure 1 we present the Speed-up ($S_p = T_1/T_p$, where T_1 is the evaluation time using the sequential Clenshaw algorithm) for the parallel Forsythe algorithm to evaluate finite series of Chebyshev polynomials of the first kind.

In Figures 2 and 3 we show the Speed-up for the parallel Clenshaw algorithm to evaluate finite series of Jacobi, Gegenbauer and Legendre polynomials, and in Figures 4 and 5 we show the same for the parallel Forsythe algorithm. The performance of the parallel Clenshaw algorithm is better than the performance of the parallel Forsythe algorithm. We observe that the time to evaluate the coefficients α_i and β_i of the triple recurrence permits us to obtain good speed-up

results, taking into account that the parallel algorithms to evaluate orthogonal series have a bigger computational complexity than the sequential ones. In the case of Chebyshev series the parallel algorithm has the same complexity as the sequential one but in this case we do not need to evaluate the coefficients.

In order to see the influence of the communication times in the parallel algorithms, we present in Figures 6,7 and 8 the speed-up without the communication process. For low degree polynomials we observe that the communication process takes more time than the evaluation when we have a high number of processors.

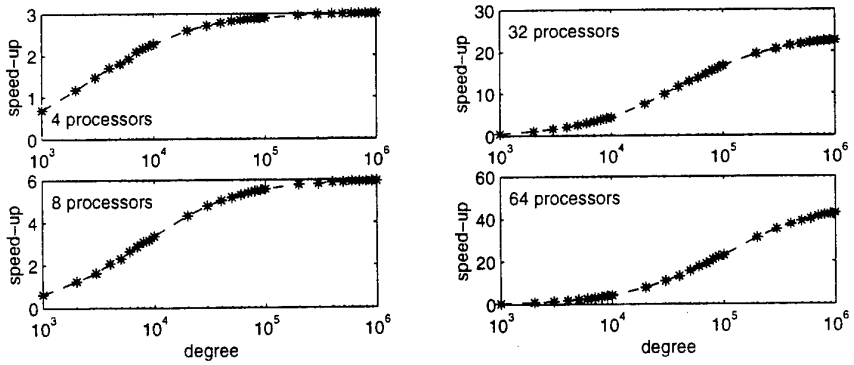


Fig. 1. Speed-up in the parallel evaluation on a CRAY T3D of several Chebyshev polynomial series using the parallel Forsythe algorithm.

Acknowledgements The author was supported in part by the Spanish Ministerio de Educación y Ciencia (DGICYT #PB95-0807). The author has performed some part of the work during a stay at the Edinburgh Parallel Computing Centre (EPCC), supported by the TRACS programme (Training and Research on Advanced Computing Systems), reference ERB–FMGE–CT95–0051, Training and Mobility of Researchers (DG–XII TMR) Programme of the European Community.

References

1. Abramowitz, M. and Stegun, I. A.: Handbook of Mathematical Functions. Dover Publications, Inc., New York (1965).
2. Barrio, R. and Sabadell, F. J.: A parallel algorithm to evaluate Chebyshev series on a message–passing environment, SIAM J. Sci. Comp. (1998), in press.
3. Barrio, R. and Sabadell, F. J.: Parallel evaluation of Chebyshev and Trigonometric series, submitted for publication (1998).
4. Clenshaw, C. W.: A note on the summation of Chebyshev series. Math. Tab. Wash. **9** (1955), 118–120.

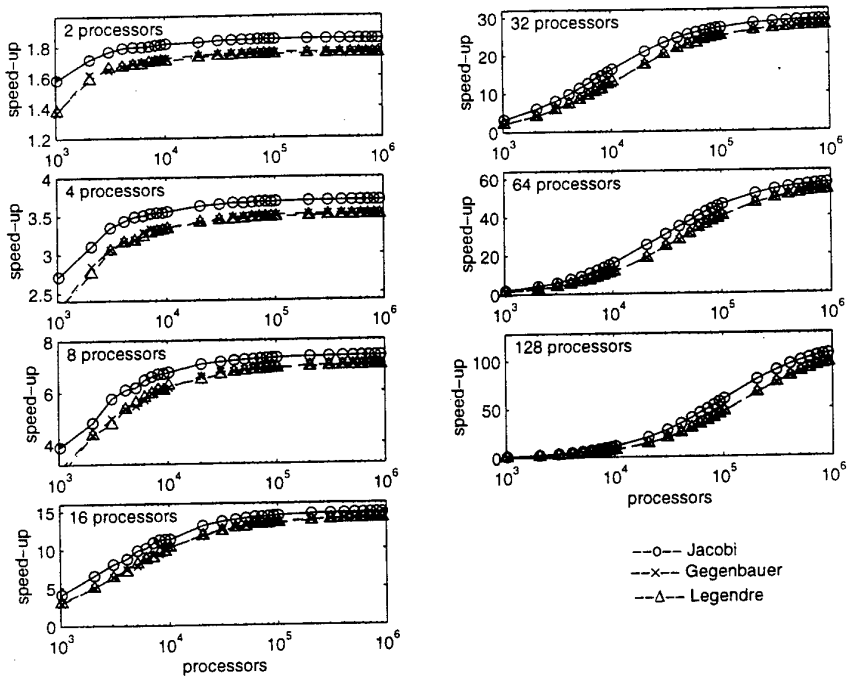


Fig. 2. Speed-up in the parallel evaluation on a CRAY T3D of several polynomials using the parallel Clenshaw algorithm.

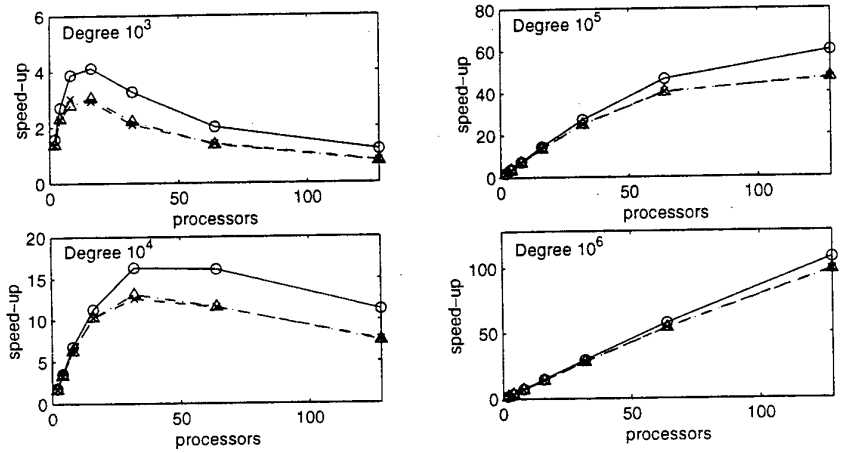


Fig. 3. Speed-up, depending on the number of processors, in the parallel evaluation on a CRAY T3D of several polynomials using the parallel Clenshaw algorithm (Jacobi polynomial series -o-, Gegenbauer polynomial series -x- and Legendre polynomial series -Δ-).

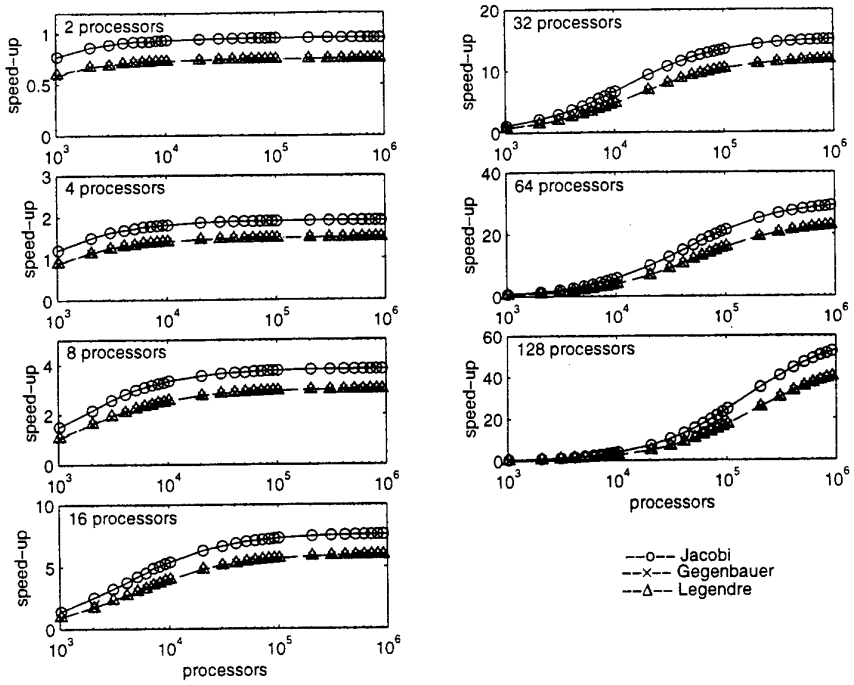


Fig. 4. Speed-up in the parallel evaluation on a CRAY T3D of several polynomials using the parallel Forsythe algorithm.

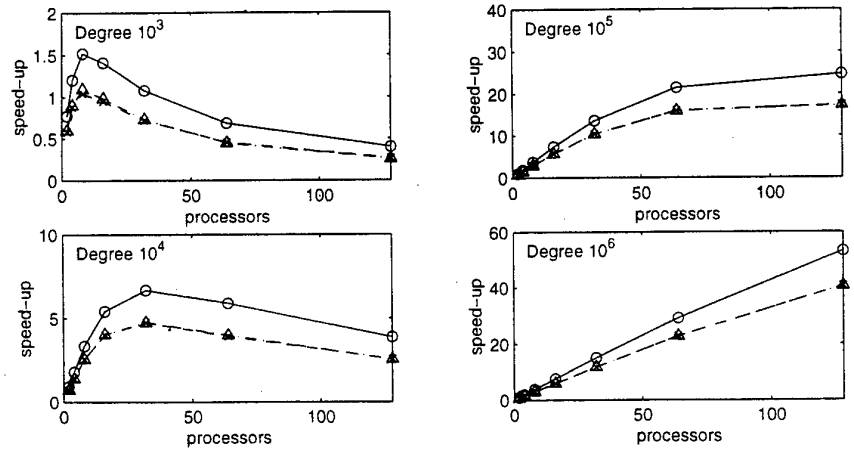


Fig. 5. Speed-up, depending on the number of processors, in the parallel evaluation on a CRAY T3D of several polynomials using the parallel Forsythe algorithm (Jacobi polynomial series -o-, Gegenbauer polynomial series -×- and Legendre polynomial series -△-).

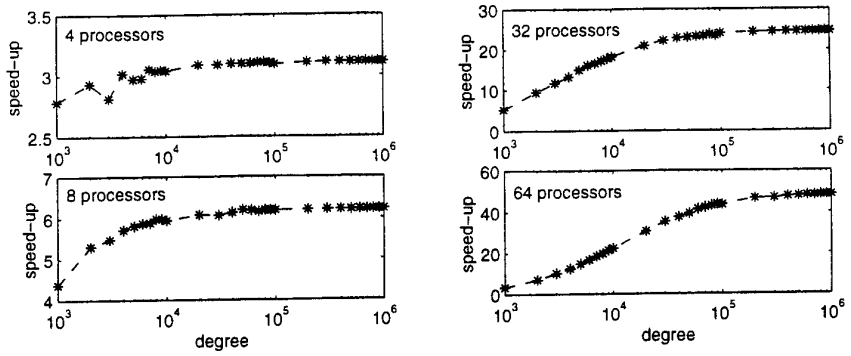


Fig. 6. Speed-up, without the communication time, in the parallel evaluation on a CRAY T3D of several Chebyshev polynomial series.

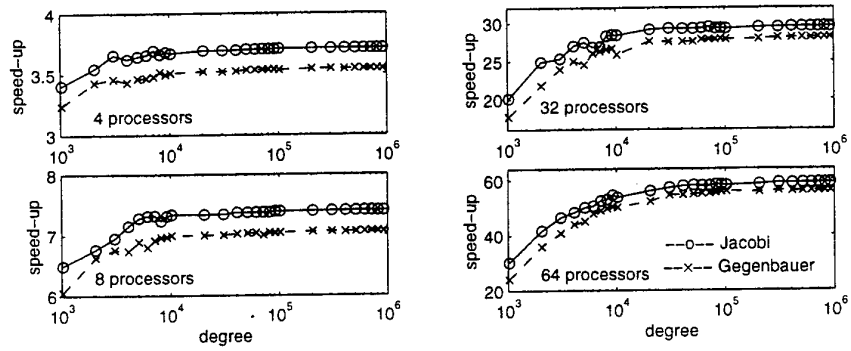


Fig. 7. Speed-up, without the communication time, in the parallel evaluation on a CRAY T3D of several polynomials using the parallel Clenshaw algorithm.

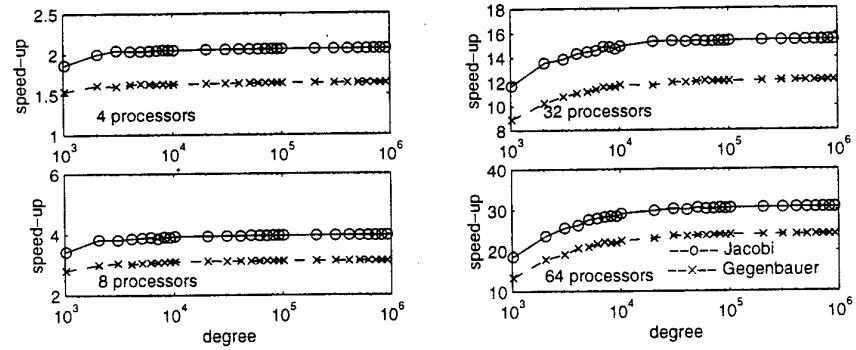


Fig. 8. Speed-up, without the communication time, in the parallel evaluation on a CRAY T3D of several polynomials using the parallel Forsythe algorithm.

5. Dongarra, J. J. and Johnsson, S. L.: Solving banded systems on a parallel processor, Parallel Computing **5** (1987), 219–246.
6. Dorn, W. S.: Generalisations of Horner's rule for polynomial evaluation, IBM J. Res. Develop. **6** (1962), 239–245. (1962).
7. Forsythe, G. E.: Generation and use of orthogonal polynomials for data fitting with a digital computer, J. SIAM **5** (1957), 74–88.
8. Johnsson, S. L.: Solving narrow banded systems on ensemble of architectures, ACM TOMS **11** (1985), 271–288.
9. Kiper, A.: Parallel polynomial evaluation by decoupling algorithm, Parallel Algorithms and Applications **9** (1996), 145–152.
10. Li, L., Hu, J. and Nakamura, T.: A simple parallel algorithm for polynomial evaluation, SIAM J. Sci. Comput. **17** (1996), 260–262.
11. Maruyama, K. : On the parallel evaluation of polynomials, IEEE Trans. Comput. **C–22** (1973), 2–5.
12. Meier, U.: A parallel partition method for solving banded systems of linear equations, Parallel Computing **2** (1985), 33–43.
13. Smith, F. J.: An algorithm for summing orthogonal polynomial series and their derivatives with applications to curve–fitting and interpolation, Math. Comp. **19** (1965), 33–36.
14. Wang, H. H.: A Parallel Method for Tridiagonal Equations, ACM TOMS **7** (1981), 170–183.

Coarse-grain parallelization of a multi-block Navier-Stokes solver on a shared memory parallel vector computer

P. Wijnandts and M.E.S. Vogels

National Aerospace Laboratory NLR, 1059 CM Amsterdam, The Netherlands

Abstract. The coarse-grain, or block-loop parallelization of the multi-block Navier-Stokes flow solver ENSOLV on a NEC SX-4, a shared memory parallel vector computer, is discussed. The performance of the parallel code was tested by running the code on ten benchmark cases, provided by the ENSOLV user group. The performance is measured in terms of speed-up, memory usage and execution cost. The results of the benchmark cases are presented. The results are compared to those of the low-level DO-loop parallelization implemented earlier. The conclusion based on the comparison of the results, is that for all benchmark cases, except the single block, the block-loop parallelization gives better performance in terms of speed-up. Although block-loop parallelization requires more memory, it gives overall less execution cost.

1 Introduction

The multi-block Navier-Stokes flow solver ENSOLV [2], [4], computes the solution of the steady 3D Euler and/or thin-layer Navier-Stokes equations in an arbitrary flow domain. The Euler and Navier-Stokes equations are given by five partial differential equations for the conservation of mass, 3D momentum and energy, extended by the perfect gas law. To solve the equations, an iterative procedure which resembles time integration is used. A number of techniques are employed to accelerate the convergence:

1. A multigrid scheme, which performs relaxations on different grid levels, is used as solution procedure. This accelerates the convergence on the finest grid level. As relaxation procedure, the explicit Runge-Kutta time stepping scheme is used;
2. The evaluation of the time step, needed for the Runge-Kutta scheme, is performed locally;
3. Implicit residual averaging with varying coefficients and enthalpy damping are used.

The solver is based on multi-block structured grids. Multigrid is applied around multi-block, i.e. on each grid level a loop on the blocks is performed. The Runge-Kutta scheme is applied on a block-by-block basis. This means that a relaxation of all blocks consists of taking one complete Runge-Kutta time step

for each block successively, keeping the flow states in the other blocks fixed. The flow solver ENSOLV is currently operational at NLR and industry.

Within the NICE[1] program, ENSOLV is being parallelized in order to reduce execution cost. Parallelization takes place on a 16-processor NEC SX-4 [9], a shared memory parallel vector computer, with a peak performance of 2 GFlop/s per processor. In [5], ten representative benchmark cases were defined by the ENSOLV user group, which constitute the benchmark for evaluating the parallelized version of the ENSOLV code. The performance of the parallel code is measured in terms of speed-up, memory usage and execution cost. At NLR, execution cost are expressed in a single number, so-called System Resource Units (SRU's). In the SRU's, the sum of all CPU-times, the amount of memory used and the time the memory is occupied, are accounted for; the formula reflects the cost price of the system elements [1]. Note that the sum of all CPU-times is always larger when parallelization is applied. If the parallelized code will result in a *reduction* in real time, by the same factor as the *increase* in memory usage, the SRU's should stay constant. A detailed explanation of the SRU formula, as used for the calculations of the SRU's reported in this document, can be found in [13].

The *Data Parallelism* strategy for parallelizing ENSOLV was chosen [8]. With this strategy, parallelism is obtained by splitting up the DO-loop's. Splitting up the DO-loop's is specifically suited for shared memory computers, such as the shared memory parallel vector machine NEC SX-4, present at NLR.

There are different levels of DO-loop parallelization, two of which are:

1. *Low-level DO-loop parallelization*, parallelization of DO-loops in individual routines. A possible problem is the fine parallel grain size; the work per loop might not be enough to overcome the parallel overhead. Also, the parallelization has to be implemented on many loops in order to achieve an acceptable parallelization percentage;
2. *Block-loop parallelization*, parallelization of the DO-loop's over the blocks in the domain. This can be considered as high-level DO-loop parallelization. It results in the largest possible grain size. A possible problem is load imbalance. The ENSOLV code uses a multigrid algorithm, which is implemented around the multi-block algorithm. The operations of the multigrid algorithm are relaxation, restriction and prolongation. The routines performing these operations all contain block-loops. Therefore, this parallelization strategy is applicable.

Earlier, ENSOLV has been parallelized using the low-level DO-loop parallelization strategy. This parallelization is described in [11]. The parallelization resulted in poor performance in terms of speed-up and execution cost, for most benchmark cases. For benchmark cases with a relatively high number of multigrid levels, combined with many small blocks in the grid, the poor performance was attributed to the large parallel overhead caused by the very fine grain size. It was decided that block-loop parallelization would be implemented. In Chapter 2,

[1] Netherlands Initiative for Computational Fluid Dynamics in Engineering with HPCN

the block-loop parallelization of ENSOLV will be described briefly. Also, the system into which the resulting parallel code, along with tools for task estimation, task allocation and speed-up estimation, was integrated, will be described. In Chapter 3, the benchmark cases will be described and remarks are made about the expected performance of the parallel code for these benchmark cases. In Chapter 4, the results of testing the block-loop parallel ENSOLV code on the benchmark cases are presented and discussed. In Chapter 5, the final conclusions are given.

2 The Parallel ENSOLV System

In this section, the block-loop parallelization of ENSOLV is described briefly. A more extensive description of the parallelization can be found in [13]. The resulting parallel code was integrated into a system including tools for task estimation, task allocation and speed-up estimation.

2.1 Block-loop parallelization of ENSOLV

Implementing block-loop parallelization, in stead of low-level DO-loop parallelization, has some consequences that need to be examined:

1. To eliminate the dependency between time integration in the blocks, the Gauss-Seidel algorithm is replaced with the Jacobi algorithm. This means that when updating the flow state of one block, the flow states from the prior Runge-Kutta time step in the adjacent blocks are used, in stead of the most recent flow states. Implementing a different solution procedure will generally change both convergence and stability, but should result in the same final solution. However, in order to allow a fast evaluation of the block-loop parallelization of ENSOLV, a simplified implementation of the Jacobi algorithm was used, resulting in a slightly different final solution (in particular near block interfaces) [3]. Results of the serial ENSOLV code using this implementation of the Jacobi algorithm, can be found in Tables 5-14;
2. A significant increase in memory usage is unavoidable; computing the blocks in parallel means that each processor needs its own scratch arrays. For all benchmark cases, except the single block benchmark case 02, the memory size is approximately doubled when run on eight processors;
3. Since blocks differ in the number of grid points, the model used, boundary conditions applied etc., a load balancing problem may occur. Implementing a load balancing, or task allocation tool will improve the load balance (Section 2.2);
4. The maximum speed-up that can be obtained, is limited to the number of blocks used, if the number of blocks is less than the number of processors. Also, if a case has one large block and many small blocks, the maximum speed-up is limited by the work load of the large block.

The block-loops were parallelized by splitting these single loops in double loops; the outer loop over the processors and the inner loop over the blocks assigned to that processor by the task allocation tool (Section 2.2). The outer loops were parallelized by inserting **odir* directives, recognizable only to the NEC Fortran compiler and therefore leading to a portable code. No message passing code is necessary, since the parallelization takes place on a shared memory computer. The NEC SX-4 preprocessor now generates the parallel code.

2.2 Integration of parallel ENSOLV

The code was integrated into a system, including tools for task estimation, task allocation and speed-up estimation. The current work was carried out by operating this system through a specific working environment, ISNaS [6], where the calculations can be started by simple drag-and-drop actions.

Task estimation Initially, the work load, or the weight for each block was set equal to the number of grid points. This is reasonable under the assumption that the work in a block is proportional only to the number of grid points, and all blocks are active in all parallel parts of the code. With ENSOLV, this assumption proved to be incorrect; if two blocks have the same number of grid points, but not the same ordering of their dimensions in the grid, their work loads can be different, due to a difference in vectorization performance.

The present task estimation tool performs (at least) one iteration of the block-loop parallel ENSOLV code, including timing-commands. The work load for each block is then set equal to the time it spends in the block-loops.

Task allocation In order to improve the load balance, a task allocation tool was implemented. This task allocation tool is a stand-alone partitioning tool, based on Ozturan algorithm [7], adapted for shared memory machines [12]. The algorithm starts from an existing partitioning of the blocks. It then re-locates blocks until a satisfying (theoretically) load balance is reached, or there is no more improvement possible.

Speed-up estimation An estimation of the maximal attainable speed-up can be made following task estimation and task allocation. An approximation of the parallel part of the code can be obtained by adding the work loads for all blocks. We can now calculate the maximal attainable speed-up using Amdahl:

$$S = \left(\frac{f \cdot \max_P W_P}{\sum_{P=1}^{N_P} W_P} + (1 - f)\right)^{-1} \qquad (1)$$

where f equals the fraction representing the parallel part, N_P equals the number of processors and W_P equals the work assigned to processor P.

3 Settings for evaluation of parallel ENSOLV

In this section, the characteristics of the benchmark cases are given. The tools in the parallel ENSOLV system are used to calculate maximal attainable speed-ups.

3.1 Characteristics of the benchmark cases

For the performance tests on the NEC SX-4, a set of test problems has been defined [5]. The characteristics of these benchmark cases can be found in Table 1. In Fig. 1, the benchmark cases are identified by configuration and number of blocks.

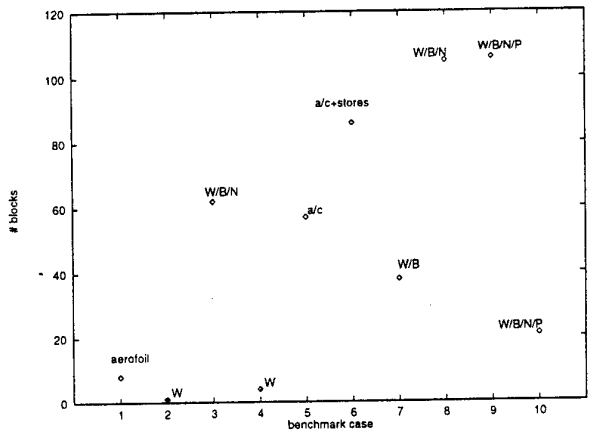


Fig. 1. Identification of benchmark cases by configuration and number of blocks

3.2 Maximal attainable speed-ups

In Tables 2 and 3, the task allocations calculated by the task allocation tool, discussed in Section 2.2, can be found. The work loads for benchmark case 05, measured by using one iteration only, were relatively small. This can lead to inaccuracies, e.g. when calculating the fraction f representing the parallel part. In order to reduce inaccuracies, the calculations for benchmark case 05 were done for the full 500 iterations. In Table 4, the maximal attainable speed-up calculated with Equation 1 can be found.

It is expected that only for benchmark case 02, a single block case, the block-loop parallelization will lead to significantly worse speed-up, compared to low-level DO-loop parallelization. For all other benchmark cases, block-loop parallelization is expected to lead to an improvement in speed-up (Fig. 2).

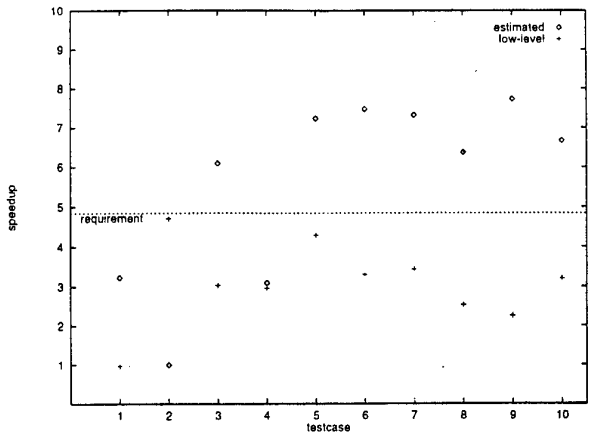


Fig. 2. Speed-up results for eight processors; estimated block-loop versus measured low-level DO-loop

4 Results

The block-loop parallelization results for all ten benchmark cases, for 1-, 4- and 8-processor runs, are shown in Tables 5-14.

In Tables 5-14, the following definitions are used:

- The *Parallelization Overhead* is defined as the ratio of the real time needed by the parallel version run on 1 processor and the real time needed by the serial version;
- The *Speed-up for* N *processors* is defined as the ratio of the real time of the serial version and the real time of the parallel version run on N processors;
- The *Memory Overhead* is defined as the ratio of the amount of memory needed by parallel ENSOLV on N processors and the amount of memory needed by the serial version.

All real time results are timings of the iteration part of the solver, output to the ENSOLV output file *OUT*.

In the following sections, the speed-up, memory usage and execution cost are compared to low-level DO-loop parallelization results. Not all the results of low-level DO-loop parallelization are listed here, the reader is referred to [11].

4.1 Speed-up results

For all benchmark cases, except the single block benchmark case 02. block-loop parallelization shows better performance in terms of speed-up, compared to low-level DO-loop parallelization.

The remaining differences in speed-up estimations and measurements are attributed to the fact that the task estimation tool uses only one iteration.

For benchmark case 05 the full 500 iterations were used, and the differences are minimal. The required speed-up of 4.8 for eight processors, defined by the ENSOLV user group, is attained by seven of the ten benchmark cases (Fig. 3).

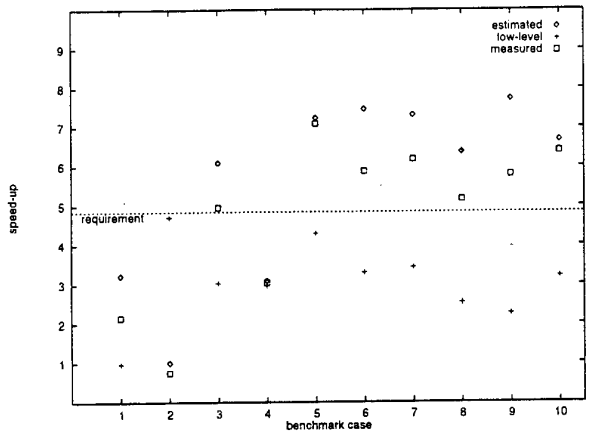


Fig. 3. Speed-up results for eight processors; measured block-loop versus estimated block-loop and measured low-level DO-loop

4.2 Memory usage

As expected, the memory usage increases considerably for all benchmark cases, except the single block benchmark case 02 (Tables 5-14). Of course, the memory usage does not further increase when the number of processors is larger than the number of blocks. Benchmark case 10 shows the largest increase in memory usage. For all benchmark cases, the memory usage was smaller than the maximal available memory on the NEC SX-4.

4.3 Execution cost

The execution cost for block-loop parallelization are considerably lower for all benchmark cases, except for the single block benchmark case 02.

For eight of the ten benchmark cases, the cost for the parallel execution of ENSOLV on eight processors are equal to or less than the cost for serial execution of ENSOLV (Fig. 4). For the large memory benchmark case 07, the cost of the parallel runs are considerably lower than the cost of the serial run. This is due to the construction of the SRU formula [13].

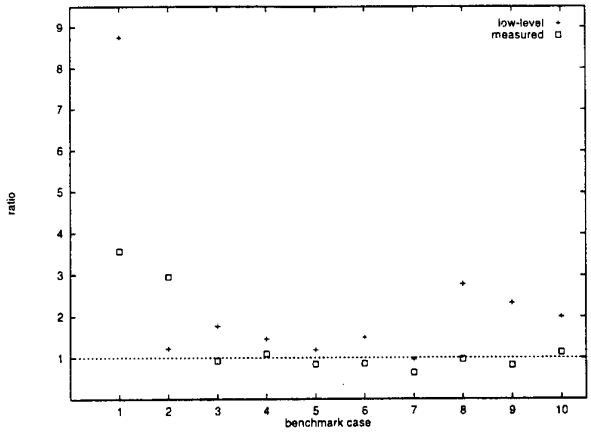


Fig. 4. Ratio of execution cost on eight processors and cost of sequential execution; measured block-loop versus measured low-level DO-loop

5 Conclusions and future work

Block-loop parallelization has been used for parallelizing the multi-block Navier-Stokes flow solver ENSOLV. The parallel code was integrated into a system, including tools for task estimation, task allocation and speed-up estimation. Future users will be able to operate this system through a specific working environment, ISNaS [6], where the calculations can be started by simple drag-and-drop actions.

The block-loop parallelized code was tested on ten benchmark cases. The performance was measured in terms of speed-up, memory usage and execution cost, and compared to the performance of the low-level DO-loop parallelized code implemented earlier.

All benchmark cases, except the single block benchmark case 02, show better performance in terms of speed-up compared to low-level DO-loop parallelization. For seven of the ten benchmark cases, the speed-up for eight processors is higher than the the user required value of 4.8.

For all benchmark cases, except the single block benchmark case 02, memory usage increases considerably when using block-loop parallelization in stead of low-level DO-loop parallelization, as was foreseen.

The block-loop parallelization gives better or comparable performance in terms of execution cost, than the low-level DO-loop parallelization, for all benchmark cases, except the single block benchmark case 02. For six of the ten benchmark cases, the execution cost for parallel runs is lower than or comparable to the execution cost for the sequential run.

Based on the results, it was decided not to implement a single parallelization approach, combining both previously applied parallelization strategies; low-level DO-loop parallelization for larger blocks, block-loop parallelization for several

smaller blocks.

References

1. Hameetman, G.: Private communications (1997)
2. Kok, J.C., Boerstoel, J.W., Kassies, A., Spekreijse, S.P.: *A robust multi-block Navier-Stokes flow solver for industrial applications*, NLR Technical Publication TP 96323 L (1996)
3. Kok, J.C.: Private communications (1998)
4. Kok, J.C.: *An industrially applicable solver for compressible, turbulent flows*, Ph.D. thesis, Delft University of Technology (1998)
5. Laban, M.: *Parallelized ENSOLV User Requirements*, NLR Technical Report TR 96353 L (1996)
6. Loeve, W., van der Ven, H., Vogels, M.E.S., Baalbergen, E.H.: *Network Middelware illustrated for enterprise enhanced operation*, NLR Technical Report TR 97224 L (1997), in CAPE'97 proceedings
7. Ozturan, C., deCougny, H.L., Shephard, M.S., Flaherty, J.E.: *Parallel adaptive mesh refinement and redistribution on distributed memory computers*, Comp. Methods Appl. Mech. Eng., Vol. 119 (1994) 123-137
8. Potma, K., Sukul, A.R.: *Preliminary ENFLOW Parallelization for the definition of a Parallelization Strategy for the NEC SX-4/16*, NLR Technical Report TR 96410 L (1996)
9. Schoenmaker, M.: *NLR SX-4/16 Vector/Parallel Supercomputer*, http://super.nlr.nl:8080/ (1998)
10. Sukul, A.R.: *Preliminary Parallelization Results of ENSOLV on the NEC SX-4*, NLR Technical Report TR 96725 L (1996)
11. Sukul, A.R.: *Predesign of ENSOLV Parallelization on the NEC SX-4*, NLR Technical Report TR 96726 L (1996)
12. van der Ven, H.: *Partitioning and parallel development of an unstructured, adaptive flow solver on the NEC SX-4*, NLR Technical Publication TP 97329 L (1997)
13. Wijnandts, P.: *Evaluation of block-loop parallelization of ENSOLV on the NEC SX-4/16*, NLR Technical Report TR 97344 L (1997)

Table 1. Characteristics of the benchmark cases (w=wing, w/b=wing-body, w/b/n=wing-body-nacelle, w/b/n/p=wing-body-nacelle-pylon, BL=Baldwin-Lomax, CS=Cebeci-Smith, JK=Johnson-King)

Case	ident	Config.	2D/3D	Blocks	Mcells	Multigrid	Euler/TLNS	Tur. Mod.	Iter.
01	RAE2822	aerofoil	2D	8	0.010	3	T(j)	BL	400
02	Delta	w	3D	1	0.369	3	T(k)	BL	200
03	AS28g	w/b/n	3D	62	1.556	2	E	-	500
04	Onera M6	w	3D	4	0.786	4	T(j)	CS	80
05	F16	a/c	3D	57	2.084	1	E	-	500
06	F16	a/c+stores	3D	86	2.084	2	E	-	360
07	VTP4	w/b	3D	38	6.636	4	T(j)	BL	100
08	VTP4	w/b/n	3D	105	1.455	3	T(j)	JK	100
09	Model 10	w/b/n/p	3D	106	2.211	3	T(i,j,k)	CS	100
10	Duprin	w/b/n/p	3D	21	0.577	2	E	-	100

Table 2. Task allocations for four processors, with W_P equal to the work load of processor P, the maximum given in bold

case	01	02	03	04	05	06	07	08	09	10
W_1	0.17	**36.20**	11.35	**29.76**	707.30	18.19	110.63	**42.50**	57.12	5.25
W_2	0.16	0	11.27	29.69	715.86	17.90	110.90	42.33	**57.69**	4.64
W_3	**0.17**	0	11.30	17.43	**726.24**	17.97	**112.44**	42.49	57.63	**5.34**
W_4	0.11	0	**11.44**	17.41	717.94	**18.21**	111.62	42.29	57.48	4.36
total	0.61	36.20	45.36	94.29	2867.34	72.27	445.59	169.61	229.92	19.59

Table 3. Task allocations for eight processors, with W_P equal to the work load of processor P, the maximum given in bold

case	01	02	03	04	05	06	07	08	09	10
W_1	**0.17**	**36.20**	5.44	**29.76**	354.89	9.00	56.12	20.50	28.79	2.57
W_2	0.16	0	5.49	29.69	**383.44**	**9.19**	55.27	20.31	**29.00**	2.70
W_3	0.07	0	**7.10**	17.43	355.95	8.96	54.23	20.45	28.89	2.08
W_4	0.10	0	5.47	17.41	352.33	8.99	56.11	21.13	28.75	2.21
W_5	0.03	0	5.54	0	353.48	8.91	**58.24**	20.32	28.61	2.59
W_6	0.01	0	5.46	0	351.00	9.07	54.72	20.27	28.54	**2.76**
W_7	0.04	0	5.41	0	366.18	9.02	56.89	**26.00**	28.47	2.49
W_8	0.03	0	5.45	0	350.07	9.13	54.01	20.63	28.87	2.19
total	0.61	36.20	45.36	94.29	2867.34	72.27	445.59	169.61	229.92	19.59

Table 4. Maximal attainable speed-ups for four and eight processors, with f the fraction representing the parallel part of the code

case	01	02	03	04	05	06	07	08	09	10
f	0.9394	0.9981	0.9908	0.9890	0.9949	0.9922	0.9932	0.9957	0.9962	0.9894
S_4	3.08	1.00	3.86	3.09	3.89	3.88	3.88	3.94	3.94	3.57
S_8	3.22	1.00	6.09	3.09	7.24	7.47	7.32	6.37	7.73	6.67

Table 5. Parallel performance for case 01, 400 iterations

#processors	sequential or parallel	execution (real) time	parallel overhead	speed-up	MFLOPS	memory usage (MB)	memory overhead	SRU
1	sequential	118	-	1.00	237	24	-	1475
1	parallel	124	1.05	0.95	226	25	1.04	1553
4	parallel	52	-	2.27	533	40	1.67	2543
8	parallel	55	-	2.15	508	54	2.25	5266

Table 6. Parallel performance for case 02, 200 iterations

#processors	sequential or parallel	execution (real) time	parallel overhead	speed-up	MFLOPS	memory usage (MB)	memory overhead	SRU
1	sequential	717	-	1.00	485	195	-	11297
1	parallel	865	1.21	0.83	436	212	1.09	12121
4	parallel	712	-	1.01	484	203	1.04	14052
8	parallel	962	-	0.75	364	212	1.09	33396

Table 7. Parallel performance for case 03, 500 iterations

#processors	sequential or parallel	execution (real) time	parallel overhead	speed-up	MFLOPS	memory usage (MB)	memory overhead	SRU
1	sequential	2256	-	1.00	666	249	-	38296
1	parallel	2327	1.03	0.97	652	266	1.07	34873
4	parallel	603	-	3.74	2488	376	1.51	33385
8	parallel	456	-	4.95	3291	465	1.87	35373

Table 8. Parallel performance for case 04, 80 iterations

#processors	sequential or parallel	execution (real) time	parallel overhead	speed-up	MFLOPS	memory usage (MB)	memory overhead	SRU
1	sequential	753	-	1.00	436	208	-	12170
1	parallel	760	1.01	0.99	433	208	1.00	12266
4	parallel	250	-	3.01	1307	398	1.91	11628
8	parallel	248	-	3.04	1324	406	1.95	13300

Table 9. Parallel performance for case 05, 500 iterations

#processors	sequential or parallel	execution (real) time	parallel overhead	speed-up	MFLOPS	memory usage (MB)	memory overhead	SRU
1	sequential	2873	-	1.00	644	241	-	48331
1	parallel	2882	1.00	1.00	643	241	1.00	48412
4	parallel	768	-	3.74	2402	357	1.48	40759
8	parallel	405	-	7.09	4541	502	2.08	40594

Table 10. Parallel performance for case 06, 360 iterations

#processors	sequential or parallel	execution (real) time	parallel overhead	speed-up	MFLOPS	memory usage (MB)	memory overhead	SRU
1	sequential	2600	-	1.00	618	282	-	45689
1	parallel	2648	1.02	0.98	608	300	1.06	40254
4	parallel	684	-	3.80	2324	434	1.54	38595
8	parallel	443	-	5.87	3584	584	2.07	39109

Table 11. Parallel performance for case 07, 100 iterations

#processors	sequential or parallel	execution (real) time	parallel overhead	speed-up	MFLOPS	memory usage (MB)	memory overhead	SRU
1	sequential	4430	-	1.00	621	859	-	129283
1	parallel	4593	1.04	0.96	617	859	1.00	130170
4	parallel	1270	-	3.49	2161	1555	1.81	91093
8	parallel	717	-	6.18	3825	1944	2.26	82421

Table 12. Parallel performance for case 08, 100 iterations

#processors	sequential or parallel	execution (real) time	parallel overhead	speed-up	MFLOPS	memory usage (MB)	memory overhead	SRU
1	sequential	1676	-	1.00	375	211	-	27197
1	parallel	1752	1.05	0.96	362	228	1.08	25539
4	parallel	560	-	2.99	1117	281	1.33	24390
8	parallel	325	-	5.16	1923	346	1.64	26005

Table 13. Parallel performance for case 09, 100 iterations

#processors	sequential or parallel	execution (real) time	parallel overhead	speed-up	MFLOPS	memory usage (MB)	memory overhead	SRU
1	sequential	2294	-	1.00	395	285	-	40452
1	parallel	2326	1.01	0.99	389	302	1.06	35627
4	parallel	681	-	3.37	1320	357	1.25	33209
8	parallel	396	-	5.79	2269	436	1.53	33046

Table 14. Parallel performance for case 10, 100 iterations

#processors	sequential or parallel	execution (real) time	parallel overhead	speed-up	MFLOPS	memory usage (MB)	memory overhead	SRU
1	sequential	198	-	1.00	570	104	-	2784
1	parallel	195	0.98	1.02	571	104	1.00	2776
4	parallel	64	-	3.09	1715	183	1.76	3028
8	parallel	31	-	6.39	3487	249	2.39	3135

This article was processed using the LaTeX macro package with LLNCS style

Using Synthetic Workloads for Parallel Task Scheduling Improvement Analysis

João Paulo Kitajima[1] and Stella Porto[2]

[1] Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Caixa Postal 702 30123-970 Belo Horizonte - MG Brazil
kita@dcc.ufmg.br
[2] Computação Aplicada e Automação
Universidade Federal Fluminense
Rua Passo da Pátria 156 24210-240 Niterói - RJ Brazil
stella@caa.uff.br

Abstract. This paper presents an experimental validation of makespan improvements of two scheduling algorithms: a greedy construction algorithm and a tabu search based algorithm. Synthetic parallel executions were performed using the scheduled graph costs. These synthetic executions were performed on a real parallel machine (IBM SP). The estimated and observed response times improvements are very similar, representing the lòw impact of system overhead on makespan improvement estimation. This guarantees a reliable cost function for static scheduling algorithms and confirms the actual better results of the tabu search metaheuristic applied to scheduling problems.

1 Introduction

Parallel applications with regular and well-known behavior, where task execution time estimates are fairly reliable, are suited to static task scheduling (in opposition to dynamic scheduling, performed during the execution of the application). This is the case of a great majority of scientific applications. For these applications, the static scheduling algorithm is executed once, before the execution of the parallel program, which is then actually run several times according to the previously obtained schedule. Consequently, even if the scheduling algorithm is a costly procedure, this cost will be amortized throughout the numerous executions of the parallel application, i.e. the obtained schedule is re-applied repeatedly.

Static task scheduling is, thus, performed based on estimated data about the parallel application and the system architecture. Therefore, realistic performance evaluation of a task scheduling algorithm can only be fully accomplished if practical results are also considered. In this sense, the present work analyzes the quality of greedy and tabu search task scheduling algorithms comparing estimated deterministic results with the actual observed makespan of several parallel synthetic applications executing on real heterogeneous parallel machines following the static schedule previously determined. The following section presents the

schedule system model. In Section 3, both the greedy and tabu search algorithms are described. In Section 4, we report the overall experimentation, including: (i) a description of the testing platform and problem instances considered during the testing phase; (ii) the most significant numerical results, and (iii) the comparative solution quality analysis according to different parameters. Section 5 presents some brief concluding remarks.

2 The Scheduling Model

A parallel application Π with a set of n tasks $T = \{t_1, \cdots, t_n\}$ and a heterogeneous multiprocessor system composed by a set of m interconnected processors $P = \{p_1, \cdots, p_m\}$ can be represented by a task precedence graph $G(\Pi)$ and an $n \times m$ matrix μ, where $\mu_{kj} = \mu(t_k, p_j)$ is the estimated execution time of a task $t_k \in T$ at processor $p_j \in P$. Each processor can run one task at a time, all tasks can be executed by any processor, and processors are said to be uniform in the sense that $\frac{\mu_{kj}}{\mu_{ki}} = \frac{\mu_{lj}}{\mu_{li}}, \forall t_k, t_l \in T, \forall p_i, p_j \in P$. This implies that processors may be ranked according to their processing speeds. In a framework with one single faster (heterogeneous) processor, the heterogeneity may be expressed by a unique parameter called *processor power ratio*, PPR, which is the ratio between the processing speed of the fastest processor and that of the remaining ones (those in the subset of homogeneous processors). Thus, an instance of our scheduling problem is characterized by the workload and parallel system models.

Given a solution s for the scheduling problem, a processor assignment function is designed as the mapping $\mathcal{A}_s : T \rightarrow P$. A task t_k is said to be assigned to processor $p_j \in P$ in solution s if $\mathcal{A}_s(t_k) = p_j$. The task scheduling problem can then be formulated as the search for an optimal assignment of the set of tasks onto that of the processors, in terms of the *makespan* $c(s)$ of the parallel application (cost of the solution s), i.e. the completion time of the last task being executed. At the end of the scheduling process, each processor ends up with an ordered list of tasks that will run on it as soon as they become executable.

3 Heuristic Task Scheduling Algorithms

We consider two algorithms in this work, namely: a greedy algorithm called DES+ MFT and a parallel tabu search algorithm, here referred as TSpar. Although both of them are heuristic, they present different fundamental characteristics. The former is a construction algorithm, which iteratively assigns tasks to processors based on heuristic criteria, taking into account the static information of the system model. On the other hand, the TSpar is a synchronous parallel implementation of a tabu search metaheuristic algorithm, which guides an aggressive local search procedure over the task scheduling solution space.

3.1 The DES+MFT Greedy Algorithm

DES+MFT stands for *Deterministic Execution Simulation with Minimum Finish Time* [4]. This algorithm iteratively schedules tasks in a partial order according

to the simulated execution of the parallel application (DES), based on the estimated task execution times, while scheduling decisions are made according to the minimum finishing time (MFT) for each "schedulable" task. Figure 1 describes the DES+MFT in a procedural scheme. In this scheme, the *clock* variable measures the evolution of the execution. At the end of this procedure, $c(s) = clock$ is the cost of the obtained solution, i.e., the makespan of the parallel application when submitted to the DES+MFT processor assignment. At each iteration, certain tasks are scheduled to processors, building an ordered list of tasks associated to each processor. This is the actual execution order if tasks were to be executed in an ideal system with estimated execution times. During this deterministic execution simulation, each task $t_k \in T$ assumes one of the following states at each time instant: ***non-executable***, ***executable***, ***executing***, ***executed***. At the same time, each processor $p_j \in P$ alternates between two different states: *free* and *busy*. A processor p_j is said to be *busy* if it has a task in the *executing* state allocated to it.

It should be noticed that DES+MFT, like most greedy algorithms, does not come back to re-evaluate the scheduling decisions taken in previous iterations. This means that besides the "look-ahead" feature, it is not capable of making changes in scheduling decisions made in previous iterations, which were based on snapshots of the simulated execution. Consequently, these scheduling decisions depend on how strongly tasks are tied through precedence relations, because they determine the order in which tasks may possibly be scheduled. Differently, the TSpar algorithm, departing from the initial solution obtained by the DES+MFT algorithm, evaluates many other possible assignments, which eventually improve the makespan of the parallel application, as we can see in the following section.

3.2 The Parallel Tabu Search Algorithm

To describe the TSpar algorithm, we first consider a general combinatorial optimization problem (P) formulated as to

$$\begin{aligned} &\text{minimize} \quad c(s) \\ &\text{subject to } s \in S, \end{aligned}$$

where S is a discrete set of feasible solutions. Local search approaches for solving problem (P) are based on search procedures in the solution space S starting from an initial solution $s_0 \in S$. At each iteration, a heuristic is used to obtain a new solution s' in the neighborhood $N(s)$ of the current solution s, through slight changes in s. A move is an atomic change which transforms the current solution, s, into one of its neighbors, say $\bar{s}$. Thus, $movevalue = c(\bar{s}) - c(s)$ is the difference between the value of the cost function after the move, $c(\bar{s})$, and the value of the cost function before the move, $c(s)$. Every feasible solution $\bar{s} \in N(s)$ is evaluated according to the cost function $c(.)$, which is eventually optimized. The current solution moves smoothly towards better neighbor solutions, enhancing the best obtained solution s^*.

Tabu search [1, 2] may be described as a higher level heuristic for solving minimization problems, designed to guide other hill-descending heuristics in order

```
DES+MFT algorithm
begin
   clock ← 0
   state(p_j) ← free ∀p_j ∈ P
   start(t_k), finish(t_k) ← 0 ∀t_k ∈ T
   while (∃t_k ∈ T | state(t_k) ≠ executed) do
   begin
      for (each t_k ∈ T | state(t_k) = executable and p_j ∈ P) do
         obtain the pair (t_l, p_i) with the minimum finish time
         if (state(p_i) = free) then
         begin
            state(t_l) ← executing
            A_s(t_l) = p_i
            state(p_i) ← busy
            start(t_l) ← clock
            finish(t_l) ← start(t_l) + μ(t_l, p_i)
         end
      Let i be such that finish(t_i) = min_{t_k∈T | state(t_k)=executing}{finish(t_k)}
      clock ← finish(t_i)
      for (each t_k ∈ T | state(t_k) = executing and finish(t_k) = clock) do
      begin
         state(t_k) ← executed
         state(A_s(t_k)) ← free
      end
   end
   c(s) ← clock
end
```

Fig. 1. DES+MFT algorithm description.

to escape from local optima. Thus, tabu search is an adaptive search technique that aims to intelligently exploring the solution space in search of good, hopefully optimal, solutions. The learning capability determines that tabu search supplies richer knowledge about the instance of the problem to be solved than that generated in other iterative algorithms. In the case of the task scheduling problem considered in this paper, the cost of a solution is given by its makespan, i.e., the overall execution time of the parallel application. The neighborhood $N(s)$ of the current solution s is the set of all solutions differing from it by only a single assignment. If $\bar{s} \in N(s)$, then there is only one task $t_i \in T$ for which $\mathcal{A}_s(t_i) \neq \mathcal{A}_{\bar{s}}(t_i)$. Each move may be characterized by a simple representation given by $(\mathcal{A}_s(t_i), t_i, p_l)$, as far as the position task t_i will occupy in the task list of processor p_l is uniquely defined. If the best move takes the current solution s to a best neighbor solution s' degenerating its cost function, i.e. $c(s') \geq c(s)$, then the reverse move must be prohibited during a certain number of iterations (tabu tenure) in order to avoid cycling. However, there are situations in which a

recently prohibited move, if applied after some iterations, will provide a better solution than the best one found by the algorithm so far, despite its prohibited status. In these cases, an *aspiration criterion* is used to override this prohibition, enabling the move to be executed. In [6] and [7] the reader will find more detailed description of the tabu search algorithm.

The promising results obtained through parallelization led to the possibility of more effectively evaluating solution quality of the proposed tabu search task scheduling algorithm using a parallel implementation. Considering both sequential and parallel implementation, solution quality was analyzed according to different parameters and strategies, which needed to fully specify the tabu search algorithm with a certain variety of application model parameters (such as task graph structures, number of tasks, serial fraction and task service demands) and system configurations (such as number of processors and architecture heterogeneity measured by the processor power ratio). It was shown that the tabu search algorithm obtained better results, i.e. shorter completion times for parallel applications, improving up to 40% the makespan obtained by the DES+MFT algorithm, which in fact is the most appropriate greedy algorithm previously published in the literature [6, 8]. We have used the MS-MP parallel version to carry out the experimentation reported here, because it has demonstrated the best speedup results in most of the studied cases [7].

4 Experimental Results

In this section, we depict some experimental results obtained from the execution of synthetic parallel programs scheduled with both the greedy and tabu search algorithms. We first present some results derived from the estimated improvement analysis of tabu search schedules over those generated by the DES+MFT, which is the initial solution for the tabu search algorithm. The performance criterium is the makespan (solution cost) estimated by both algorithms. In the following, we describe *ANDES* [3], a framework for performance evaluation using parallel program models and synthetic programs. Finally, using this framework, we compare execution times of synthetic parallel programs scheduled by DES+MFT and TSpar algorithms.

4.1 Estimated Performance Analysis

DES+MFT and TSpar scheduling algorithms were implemented using ANSI C and PVM (*Parallel Virtual Machine*) [9]. The schedule quality is estimated based on the computed makespan. In other words, the makespan represents the schedule cost, $c(.)$, which is to be minimized.

One of the main goals is to achieve makespan reduction when changing from the schedule produced by DES+MFT to the one produced by TSpar. Thus, solution quality is measured by relative cost reduction, $\mathcal{R}$, computed as

$$\mathcal{R} = \frac{c(s_0) - c(s^*)}{c(s_0)}$$

where s_0 is the initial solution obtained by the greedy algorithm DES+MFT and s^* is the best solution found by the TSpar algorithm.

In [6], relative cost reduction values of up to 30% were obtained considering applications modeled by diamond-shaped precedence graphs. In [8], new results were presented considering other structures for the parallel applications. Part of the *ANDES* benchmark was then used: other types of diamond-shaped graphs (`Diamond3` and `Diamond4`), iterative graphs (`FFT` and `PDE2`), divide-and-conquer strategies (`Divconq`), typical matrix computation structures (`Gauss`), and master-slave models (`MS3`). We can summarize the following results of these above experiments:

- A parallel application is said to be *serialized* by a certain processor assignment algorithm when all of its tasks are scheduled to one unique processor. When the serial fraction (F_s) and/or the processor power ratio (PPR) are very high, the best solution is usually obtained through the serialization of the application over the heterogeneous processor, which has greater processing capacity. This seems to be clear if we imagine two extreme cases: $F_s = 1$ or $PPR \longrightarrow \infty$. In the first case, we face a totally serial application, which must be executed on the heterogeneous processor (F_s corresponds to the serial fraction defined as the fraction of the total parallel execution time when just one task is executing even if infinite processors were available). In the latter case, the heterogeneous processor is able to execute any task in infinitesimal time, consequently serialization determines again the best performance.

  In certain circumstances, serialization will be performed by the DES+MFT algorithm, when there is still available parallelism to be explored in the parallel application. In these cases, the tabu search algorithm will start from a serialized initial schedule, and more easily will be capable of finding different assignments which greatly reduce the overall makespan of the application, augmenting the relative cost reduction.

  For very low and very high PPR values low or null makespan improvements are obtained. A low PPR value means low heterogeneity degree, and, in this case, the greedy algorithm improvements are sufficient (it is suitable for homogeneous configurations). On the other end of the heterogeneity range, very high PPR values mean that *serialization* on the very fast processor is the best solution. In these cases both the DES+MFT and TSpar algorithms are able of serializing the application, so makespan improvements are not observed;
- Between the two extremes of the PPR value range, we find a mountain-like peak of improvements, culminating with a PPR that gives the best relative performance achieved by the TSpar algorithm. This point is referred as the PPR_{peak} point. The PPR_{peak} point is highly dependent on the shape of the input task graph. Groups of similar task graphs have a similar behavior. For example, diamond-shaped graphs present a low PPR_{peak} (around 5). On the other hand, iterative graphs produce a more smooth improvement

curve, with higher PPR_{peak} (around 20 or 30), depending on the size of the task graph;
- Not only the structure of the task graph is critical in the relative quality improvement analysis. The number of processors available for scheduling assignments influence the results. The relationship between solution quality improvements and the number of processors is variable depending on the structure of the task graph. On one hand, the greater the number of processors we have, the less heterogeneous the system becomes and thus lower relative cost reduction is achieved. However, a greater number of processors also represents more available parallelism and therefore a greater number of different scheduling possibilities arise.

Figure 3 presents some estimated relative cost reduction values computed between DES+MFT and TSpar algorithms. In [8], Porto *et al.* measured improvements for discrete values of PPR (2, 5, 10, 20, ..., depending on the input). Figure 2 presents a more detailed experiment, with a fine variation of PPR values and number of processors, considering the `Diamond3` benchmark with 66 tasks.

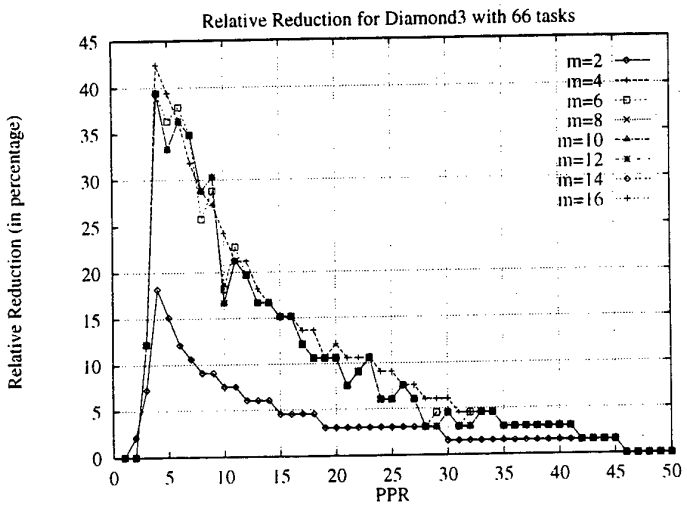


Fig. 2. Detailed relative cost reduction $\mathcal{R}$ versus PPR for `Diamond3` graph (m corresponds to the number of processors to which the tasks are scheduled).

4.2 The Experimental Framework

The *ANDES* Environment - *ANDES* [8] is a PVM-based parallel tool that supports performance evaluation of parallel programs at the prediction level.

ANDES considers the existing complex overheads of parallel computers. This is achieved through the use of *synthetic parallel executions* directly on the parallel machine. In a synthetic parallel execution, the resources of the parallel computer are used in a controlled way, but no code is generated. All the steps from the interpretation of the parallel program graph-based and of the parallel machine models to the synthetic execution on the target parallel machine are automatically managed by *ANDES*. ANDES finally computes performance metrics along the execution of that workload implemented according to mapping and/or scheduling strategies. Synthetic execution was chosen as the performance technique due to the easy control of parameters as well as the possibility of using a real envionment. The idea is to conjugate the best of model-based approaches with the best of realistic parallel executions. *ANDES* has been used to refine analytical and simulation analysis. With the current high availability of parallel systems, the results of *ANDES* have been proved to be precise and useful.

The Parallel System – *ANDES* along with the synthetic parallel programs were executed on an IBM SP multicomputer composed of 32 RS6000 RISC microprocessors with 64 megabytes of RAM. The processors are interconnected by a high-speed switch (bidirectional with nominal speed of 80 megabytes per second).

The Benchmarks – In order to compare estimated and observed improvements of the overall execution times of real parallel synthetic programs, we have used the following benchmark (part of the *ANDES* package): (i) `Diamond3` with 66 tasks; (ii) `FFT` with 194 tasks; (iii)`Gauss` with 192 tasks; and (iv) `Divconq` with 46 tasks.

This benchmark picks representative task graphs from the ones studied in [8]. Small and larger task graphs are used. The TSpar was executed using 4 processors of the IBM SP. The estimated quality of both TSpar and DES+MFT algorithms is computed using a conventional C procedure for computing the makespan of the task graphs, detailed in Figure 4 (very similar to the DES+ MFT description). The final value of *clock* is the actual makespan. Each graph of the benchmark is scheduled to 2, 4, 8, and 16 processors.

The generated schedules are read by *ANDES* which generates the synthetic load to be interpreted by *ANDES-Synth*, the synthetic execution kernel. Synthetic loads are then executed according to the given schedules.

In order to simulate heterogeneity, the size of synthetic loops corresponding to tasks allocated to the faster processor are reduced by a factor corresponding to the PPR itself. Thus, a PPR of 2 means that loops to be executed on the heterogeneous processor are reduced by half. The scheduling algorithms consider communications with zero overhead. This corresponds in *ANDES* to communications of a single byte (in the IBM SP machine, such message transmitted through the switch determines a latency of around 47.03 microseconds [5]).

Preliminary experiments were performed on an idle machine. The standard deviation was always under 1% for 10 consecutive executions. Considering this low degree of variability, we have performed measures using a sample of size 5.

4.3 Results and Analysis

Figures 5, 6, 7, and 8 present, in the same graphic, estimated and measured relative cost reduction values. The chosen PPR value range includes, for all graphics, the higher relative cost reduction values achieved by TSpar. Differences between estimated and observed improvements are under 5% for all experiments.

Our results demonstrated by the similarity between estimated and observed relative cost reduction values that the makespan computation used in both scheduling algorithms is in fact reliable. This computation is completely deterministic. On the other hand, the observed execution times are definitely nondeterministic due to the overhead from the operating system and the communication subsystem. However, the execution times presented very low variability. Therefore, this overhead does not influence significantly the experimental execution times, i.e. the makespan algorithm shows itself to be very useful to the static scheduling decisions based on estimated data. Although intuitive, this conclusion is not obvious and experiments were necessary to validate it.

Taking into account a precise makespan computation, one important consequence is that tabu search improvements are real and significant. This was foreseen from previous work, based on the estimated relative cost reduction values between DES+MFT and TSpar algorithms. In this paper, we demonstrate that these improvements also occur in more realistic execution environments.

Another interesting result is that the PPR_{peak} is not always the same. As a matter of fact, there is a range of PPR values where the best relative cost reduction varies. This irregular behavior occurs due to the irregular search through the solution space performed by the tabu search algorithm, which depends on different heuristic parameters such as tabu list size, number of iterations without improvements, and aspiration criteria. Metaheuristics, such as tabu search, frequently depend on a fine tuning stage, where parameters are tested and calibrated. After this step, they remain unchanged, and in some test cases they are not always set to achieve the best results.

Finally, *ANDES* has been proven to be a useful tool in the validation of scheduling algorithms. The direct combination of both scheduling algorithms and the synthetic execution runtime system provided an environment where response time measurements could be quickly obtained.

5 Final Remarks

This paper presents an experimental validation of makespan improvements of two scheduling algorithms: a greedy construction algorithm and a tabu search based algorithm. Synthetic parallel executions were performed given data on task execution times, task precedence relations, and task scheduling. These synthetic executions were performed on a real parallel machine (IBM SP). The estimated and observed response times improvements are very similar, representing the low impact of system overhead on makespan improvement estimation. This guarantees a reliable cost function for static scheduling algorithms and confirms

the actual better results of the tabu search metaheuristic applied to scheduling problems.

References

1. F. GLOVER and M. LAGUNA, "Tabu Search", Chapter 3 in *Modern Heuristic Techniques for Combinatorial Problems* (C.R. Reeves, ed.), 70–150, Blackwell Scientific Publications, Oxford, 1992.
2. F. GLOVER, E. TAILLARD, and D. DE WERRA, "A User's Guide to Tabu Search", *Annals of Operations Research* 41 (1993), 3–28.
3. J.P. KITAJIMA, B. PLATEAU, P.BOUVRY, and D. TRYSTRAM, "A method and a tool for performance evaluation. A case study: Evaluating mapping strategies", *Proceedings of the 1994 Cray Users Group Meeting*. Tours, 1994.
4. D.A. MENASCÉ and S.C.S. PORTO, "Processor Assignment in Heterogeneous Parallel Architectures", *Proceedings of the IEEE International Parallel Processing Symposium*, 186–191, Beverly Hills, 1992.
5. J. MIGUEL, A. ARRUABARRENA, R. BEIVIDE, and J. A. GREGORIO, "Assessing the performance of the new IBM SP2 communication subsystem", *IEEE Parallel & Distributed Technology* 4(1996), 12–22.
6. S.C.S. PORTO and C.C. RIBEIRO, "A Tabu Search Approach to Task Scheduling on Heterogeneous Processors under Precedence Constraints", *International Journal of High-Speed Computing* 7 (1995), 45–71.
7. S.C.S. PORTO and C.C. RIBEIRO, "Parallel Tabu Search Message-Passing Synchronous Strategies for Task Scheduling under Precedence Constraints", *Journal of Heuristics* 1 (1996), 207–233.
8. S.C.S. PORTO, J.P.W. KITAJIMA, and C.C. RIBEIRO, "Performance Evaluation of a Parallel Tabu Search Scheduling Algorithm", *Solving Combinatorial Problems in Parallel* (joint workshop with the *International Parallel Processing Symposium'97*), April 1-5 1997, Geneva.
9. V. S. SUNDERAM, G. A. GEIST, J. DONGARRA, and R. MANCHEK, "The PVM concurrent computing system: evolution, experiences, and trends", *Parallel Computing* 20(1994), 531–546.

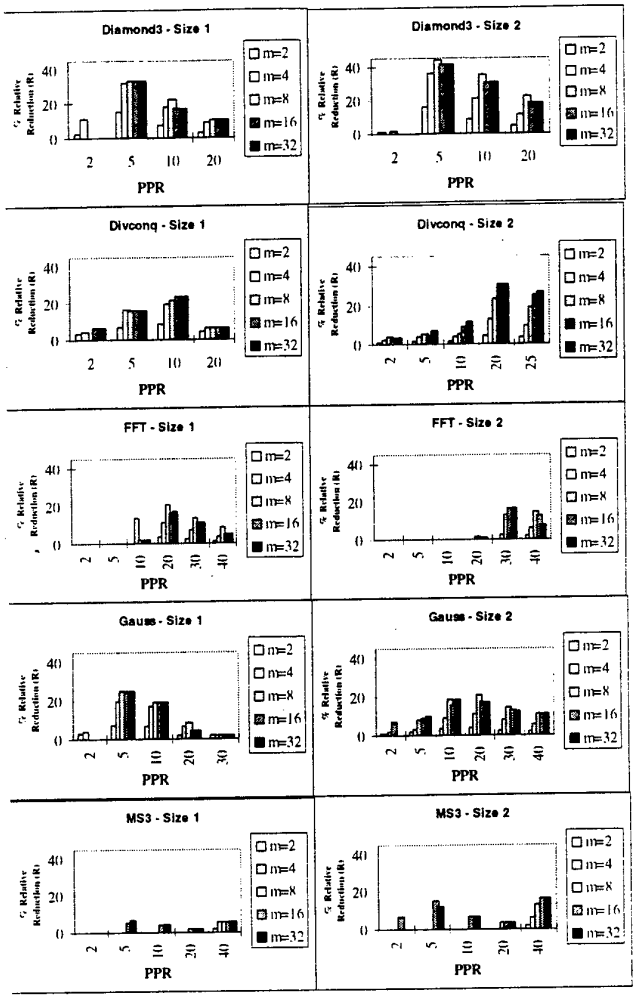


Fig. 3. Relative cost reduction $\mathcal{R}$ versus PPR for two different sizes of `Diamond3`, `Divconq`, `FFT`, `Gauss`, and `MS3` graphs (m corresponds to the number of processors to which the tasks are scheduled).

makespan computation algorithm
begin
Let $s = (\mathcal{A}_s(t_1), \ldots, \mathcal{A}_s(t_n))$ be a feasible solution for the scheduling problem, i.e.,
for every $k = 1, \ldots, n$, $\mathcal{A}_s(t_k) = p_j$ for some $p_j \in P$
$clock \leftarrow 0$
$state(p_j) \leftarrow$ free $\forall p_j \in P$
$start(t_k), finish(t_k) \leftarrow 0\ \forall t_k \in T$
while $(\exists t_k \in T \mid state(t_k) \neq$ executed) **do**
begin
for (each $t_k \in T \mid state(t_k) =$ executable) **do**
if $(state(\mathcal{A}_s(t_k)) =$ free) **then**
begin
$state(t_k) \leftarrow$ executing
$state(\mathcal{A}_s(t_k)) \leftarrow$ busy
$start(t_k) \leftarrow clock$
$finish(t_k) \leftarrow start(t_k) + \mu(t_k, \mathcal{A}_s(t_k))$
end
Let i be such that $finish(t_i) = \min_{t_k \in T \mid state(t_k)=\text{executing}}\{finish(t_k)\}$
$clock \leftarrow finish(t_i)$
for (each $t_k \in T \mid state(t_k) =$ executing and $finish(t_k) = clock$) **do**
begin
$state(t_k) \leftarrow$ executed
$state(\mathcal{A}_s(t_k)) \leftarrow$ free
end
end
$c(s) \leftarrow clock$
end

Fig. 4. Computation of the makespan of a given schedule.

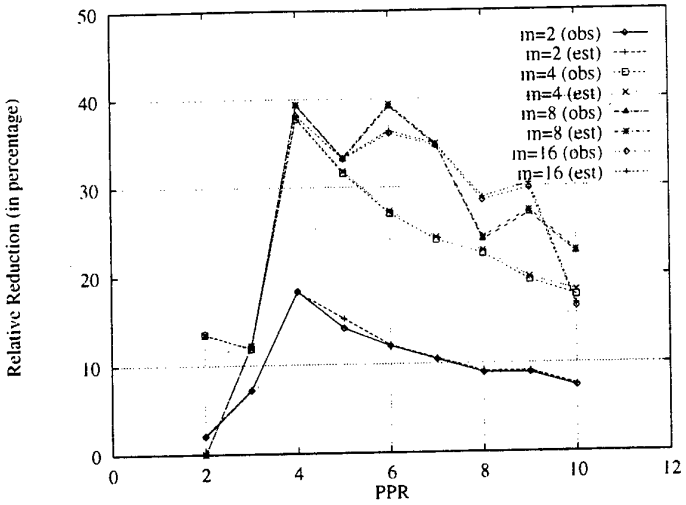


Fig. 5. Estimated (est) and observed (obs) relative cost reduction $\mathcal{R}$ versus PPR for `Diamond3` graph (m corresponds to the number of processors on which the tasks are scheduled).

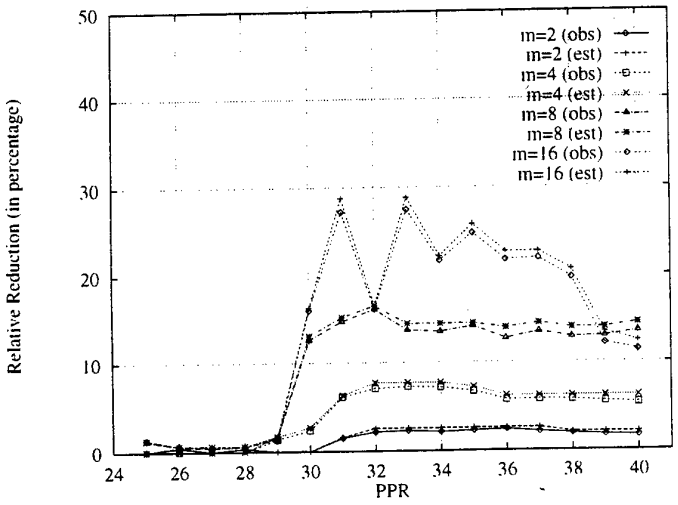


Fig. 6. Estimated (est) and observed (obs) relative cost reduction $\mathcal{R}$ versus PPR for `FFT` graph (m corresponds to the number of processors on which the tasks are scheduled).

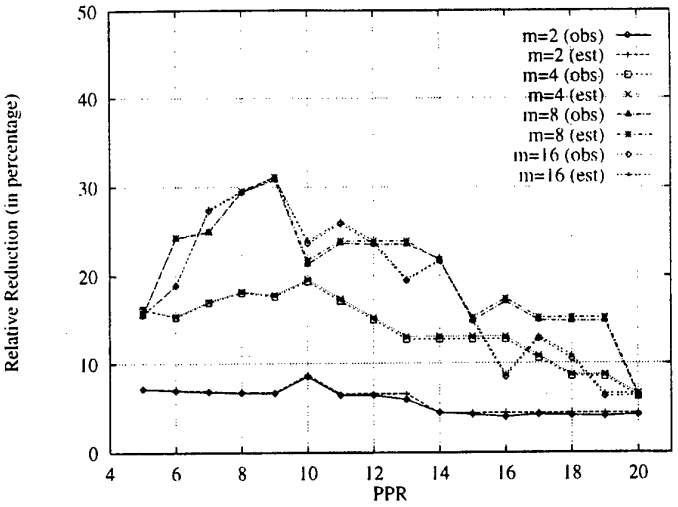


Fig. 7. Estimated (est) and observed (obs) relative cost reduction $\mathcal{R}$ versus PPR for `Divconq` graph (m corresponds to the number of processors on which the tasks are scheduled).

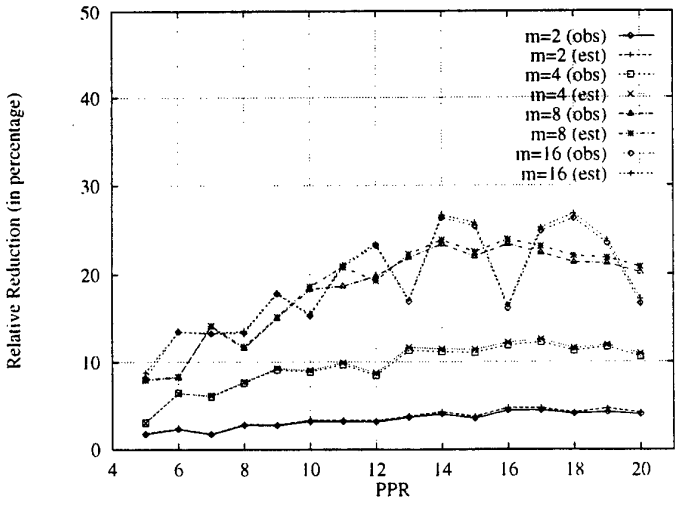


Fig. 8. Estimated (est) and observed (obs) relative cost reduction $\mathcal{R}$ versus PPR for `Gauss` graph (m corresponds to the number of processors on which the tasks are scheduled).

Influence of the Discretization Scheme on the Parallel Efficiency of a Code for the Modelling of a Utility Boiler

Pedro Jorge Coelho

Instituto Superior Técnico, Technical University of Lisbon
Mechanical Engineering Department
Av. Rovisco Pais, 1096 Lisboa Codex, Portugal

Abstract. A code for the simulation of the turbulent reactive flow with heat transfer in a utility boiler has been parallelized using MPI. This paper reports a comparison of the parallel efficiency of the code using the hybrid central differences/upwind and the MUSCL schemes for the discretization of the convective terms of the governing equations. The results were obtained using a Cray T3D and a number of processors in the range 1 - 128. It is shown that higher efficiencies are obtained using the MUSCL scheme and that the least efficient tasks are the solution of the pressure correction equation and the radiative heat transfer calculation.

Keywords: Parallel Computing; Discretization Schemes; Computational Fluid Dynamics; Radiation; Boilers

1 Introduction

The numerical simulation of the physical phenomena that take place in the combustion chamber of a utility boiler is a difficult task due to the complexity of those phenomena (turbulence, combustion, radiation) and to the range of geometrical length scales which spans fours or five orders of magnitude [1]. As a consequence, such a simulation is quite demanding as far as the computational resources are concerned. Therefore, parallel computing can be very useful in this field.

The mathematical modelling of a utility boiler is often based on the numerical solution of the equations governing conservation of mass, momentum and energy, and transport equations for scalar quantities describing turbulence and combustion. These equations are solved in an Eulerian framework and their numerical discretization yields convective terms which express the flux of a dependent variable across the faces of the control volumes over which the discretization is carried out. Many dicretization schemes for the convective terms have been proposed along the years and this issue has been one of the most important topics in computational fluid dynamics research.

The hybrid central differences/upwind scheme has been one the most popular ones, especially in incompressible flows. However, it reverts to the first order upwind scheme whenever the absolute value of the local Peclet number is higher than two, which may be the case in most of the flow regions. This yields poor accuracy and numerical diffusion errors. These can only be overcome using a fine grid which enables a reduction of the local Peclet number, and will ultimately revert the scheme to the second order accurate central differences scheme. However, this often requires a grid too fine, and there is nowadays general consensus that the hybrid scheme should not be used (see, e.g., [2]). Moreover, some leading journals presently request that solution methods must be at least second order accurate in space. Alternative discretization schemes, such as the skew upwind, second order upwind and QUICK, are more accurate but may have stability and/or boundedness problems. Remedies to overcome these limitations have been proposed more recently and there are presently several schemes available which are stable, bounded and at least second order accurate (see, e.g., [3 - 9]).

Several high resolution schemes have been incorporated in the code presented in [10] for the calculation of laminar or turbulent incompressible fluid flows in two or three-dimensional geometries. Several modules were coupled to this code enabling the modelling of combustion, radiation and pollutants formation. In this work, the code was applied to the simulation of a utility boiler, and a comparison of the efficiency obtained using the hybrid and the MUSCL ([11]) schemes is presented. The mathematical model and the parallel implementation are described in the next two sections. Then, the results are presented and discussed, and the conclusions are summarized in the last section.

2 The Mathematical Model

2.1 Main features of the model

The mathematical model is based on the numerical solution of the density weighted averaged form of the equations governing conservation of mass, momentum and energy, and transport equations for scalar quantities. Only a brief description of the reactive fluid flow model is given below. Further details may be found in [12].

The Reynolds stresses and the turbulent scalar fluxes are determined by means of the k-ε eddy viscosity/diffusivity model which comprises transport equations for the turbulent kinetic energy and its dissipation rate. Standard values are assigned to all the constants of the model.

Combustion modelling is based on the conserved scalar/probability density function approach. A chemical equilibrium code is used to obtain the relationship between instantaneous values of the mixture fraction and the density and chemical species concentrations. The calculation of the mean values of these quantities requires an integration of the instantaneous values weighted by the assumed probability density

function over the mixture fraction range. These calculations are performed *a priori* and stored in tabular form.

The discrete ordinates method [13] is used to calculate the radiative heat transfer in the combustion chamber. The S_4 approximation, the level symmetric quadrature satisfying sequential odd moments [14] and the step scheme are employed. The radiant superheaters which are suspended from the top of the combustion chamber are simulated as baffles without thickness as reported in [15]. The radiative properties of the medium are calculated using the weighted sum of grey gases model.

The governing equations are discretized over a Cartesian, non-staggered grid using a finite volume/finite difference method. The convective terms are discretized using either the hybrid or the MUSCL schemes. The solution algorithm is based on the SIMPLE method. The algebraic sets of discretized equations are solved using the Gauss-Seidel line-by-line iterative procedure, except the pressure correction equation which is solved using a preconditioned conjugate gradient method.

2.2 Discretization of the convective terms

The discretized equation for a dependent variable ϕ at grid node P may be written in the following compact form:

$$a_P \phi_P = \sum_i a_i \phi_i + b \qquad (1)$$

where the coeficients a_i denote combined convection/diffusion coefficients and b is a source term. The summation runs over all the neighbours of grid node P (east, west, north, south, front and back). Derivation of this discretized equation may be found, e.g., in [16]. If the convective terms are computed by means of the hybrid upwind/central differences method, then the system of equations (1) is diagonally dominant and can be solved using any conventional iterative solution technique. If a higher order discretization scheme is used, the system of equations may still have a diagonally dominant matrix of coefficients provided that the terms are rearranged using a deferred correction technique [17]. In this case, equation (1) is written as:

$$a_P^U \phi_P = \sum_i a_i^U \phi_i + b + \sum_j C_j \left(\phi_j^U - \phi_j\right) \qquad (2)$$

where the superscript U means that the upwind scheme is used to compute the corresponding variable or coefficient, and Cj is the convective flux at cell face j. The last term on the right hand side of the equation is the contribution to the source term due to the deferred correction procedure.

The high order schemes were incorporated in the code using the normalized variable and space formulation methodology [18]. According to this formulation, denoting by U, C, and D the upstream, central and downstream grid nodes surrounding the control volume face f (see Figure 1), the following normalized variables are defined:

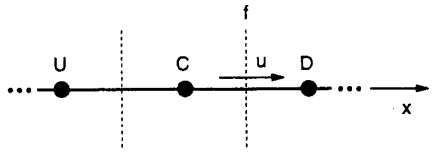


Figure 1 - Interpolation grid nodes involved in the calculation of ϕ_f.

$$\tilde{\phi} = \frac{\phi - \phi_U}{\phi_D - \phi_U} \tag{3}$$

$$\tilde{x} = \frac{x - x_U}{x_D - x_U} \tag{4}$$

where x is the coordinate along the direction of these nodes. The upwind scheme yields:

$$\tilde{\phi}_f = \tilde{\phi}_C \tag{5}$$

while the MUSCL scheme is given by:

$$\tilde{\phi}_f = (2\,\tilde{x}_f - \tilde{x}_C)\,\tilde{\phi}_C / \tilde{x}_C \quad \text{if} \quad 0 < \tilde{\phi}_C < \tilde{x}_C/2 \tag{6a}$$

$$\tilde{\phi}_f = \tilde{x}_f - \tilde{x}_C + \tilde{\phi}_C \quad \text{if} \quad \tilde{x}_C/2 < \tilde{\phi}_C < 1 + \tilde{x}_C - \tilde{x}_f \tag{6b}$$

$$\tilde{\phi}_f = 1 \quad \text{if} \quad 1 + \tilde{x}_C - \tilde{x}_f < \tilde{\phi}_C < 1 \tag{6c}$$

$$\tilde{\phi}_f = \tilde{\phi}_C \quad \text{elsewhere} \tag{6d}$$

3 Parallel Implementation

The parallel implementation is based on a domain decomposition strategy and the communications among the processors are accomplished using MPI. This standard is now widely available and the code is therefore easily portable across hardware ranging from workstation clusters, through shared memory modestly parallel servers to massively parallel systems. Within the domain decomposition approach the computational domain is split up into non-overlapping subdomains, and each subdomain is assigned to a processor. Each processor deals with a subdomain and communicates and/ synchronizes its actions with those of other processors by exchanging messages.

The calculation of the coefficients of the discretized equations in a control volume requires the knowledge of the values of the dependent variables at one or two neighbouring control volumes along each direction. Only one neighbour is involved if the hybrid scheme is used, while two neighbours are involved if the MUSCL scheme

is employed. In the case of control volumes on the boundary of a subdomain, the neighbours lie on a subdomain assigned to a different processor. In distributed memory computers a processor has only in its memory the data of the subdomain assigned to that processor. This implies that it must exchange data with the neighbouring subdomains.

Data transfer between neighbouring subdomains is simplified by the use of a buffer of halo points around the rectangular subdomain assigned to every processor. Hence, two planes of auxiliary points are added to each subdomain boundary, which store data calculated in neighbouring subdomains. These data is periodically exchanged between neighbouring subdomains to ensure the correct coupling of the local solutions into the global solution. This halo data transfer between neighbouring processors is achieved by a pair-wise exchange of data. This transfer proceeds in parallel and it will be referred to as local communications. Local communication of a dependent variable governed by a conservation or transport equation is performed just before the end of each sweep (inner iteration) of the Gauss-Seidel procedure for that variable. Local communication of the mean temperature, density, specific heat and effective viscosity is performed after the update of these quantities, i.e., once per outer iteration. Besides the local communications, the processors need to communicate global data such as the values of the residuals which need to be accumulated, or maximum or minimum values determined, or values broadcast. These data exchange are referred to as global communications and are available in standard message passing interfaces.

While the parallelization of the fluid flow equations solver has been widely addressed in the literature, the parallelization of the radiation model has received little attention. The method employed here is described in detail in [19-20] and uses the spatial domain decomposition method for the parallelization of the discrete ordinates method. It has been found that this method is not as efficient as the angular decomposition method, since the convergence rate of the radiative calculations is adversely influenced by the decomposition of the domain, dropping fast as the number of processors increases. However, the compatibility with the domain decomposition technique used in parallel computational fluid dynamics (CFD) codes favours the use of the spatial domain decomposition method for the radiation in the case of coupled fluid flow/heat transfer problems.

4 Results and Discussion

The code was applied to the simulation of the physical phenomena taking place in the combustion chamber of a power station boiler of the Portuguese Electricity Utility. It is a natural circulation drum fuel-oil fired boiler with a pressurized combustion chamber, parallel passages by the convection chamber and preheating. The boiler is fired from three levels of four burners each, placed on the front wall. Vaporization of the fuel is assumed to occur instantaneously. At maximum capacity (771 ton/h at 167 bar and 545°C) the output power is 250 MWe. This boiler has been extensively

investigated in the past, both experimentally and numerically (see, e.g., [1, 21-23]), and therefore no predictions are shown in this paper which is concentrated only on the parallel performance of the code.

The calculations were performed using the Cray T3D of the University of Edinburgh in U.K. It comprises 256 nodes each with two processing units. Each processing element consists of a DEC Alpha 21064 processor running at 150MHz and delivering 150 64-bit Mflop/s. The peak performance of the 512 processing elements is 76.8 Gflop/s.

Jobs running in the computer used in the present calculations and using less than 64 processors are restricted to a maximum of 30 minutes. Therefore, to allow a comparison between runs with different number of processors the initial calculations, summarized in tables 1 and 2, were carried out for a fixed number of iterations (30 for the MUSCL discretization scheme and 80 for the hybrid scheme). They were obtained using a grid with 64,000 grid nodes (20x40x80). Radiation is not accounted for in this case. For a given number of processors different partitions of the computational domain were tried, yielding slightly different results. The influence of the partition on the attained efficiency is discussed in [24]. Only the results for the best partitions, as far as the efficiency is concerned, are shown in tables 1 and 2.

The parallel performance of the code is examined by means of the speedup, S, defined as the ratio of the execution time of the parallel code on one processor to the execution time on n_p processors (t_{total}), and the efficiency, ε, defined as the ratio of the speedup to the number of processors. The results obtained show that the highest speedups are obtained when the MUSCL discretization scheme is employed. For example when 128 processors are used speedups of 95.1 and 77.4 are achieved using the MUSCL and the hybrid discretization schemes, respectively. As the number of processors increases, so does the speedup of the MUSCL calculations compared to the hybrid calculations. For n_p=2 we have S(MUSCL)/S(hybrid)=1.04 while for n_p=128 that ratio is 1.23. There are two opposite trends responsible for this behaviour. In fact, there is more data to communicate among the processors when the MUSCL scheme is used, because there are two planes of grid nodes in the halo region compared to only one when the hybrid scheme is employed. However, the calculation time is significantly higher for the MUSCL scheme since the computation of the coefficients of the discretized equations is more involved. Overall, the ratio of the communications to the total time is larger for the hybrid scheme, yielding smaller speedups.

The calculations were divided into five tasks, for analysis purposes, and their partial efficiency is shown in tables 1 and 2. These tasks are: i) the solution of the momentum equations, including the calculation of the convective, diffusive and source terms, the incorporation of the boundary conditions, the calculation of the residuals, the solution of the algebraic sets of equations and the associated communications among processors; ii) the solution of the pressure correction equation and the correction of the velocity and pressure fields according to the SIMPLE algorithm; iii) the solution of the transport equations for the turbulent kinetic energy and its dissipation rate; iv) the solution of other scalar transport equations, namely

Table 1. Parallel performance of the code for the first 80 iterations using the hybrid discretization scheme.

n_p	1	2	4	8	16	32	64	128
Partition	1x1x1	1x2x1	1x4x1	1x8x1	2x4x2	2x4x4	2x4x8	2x8x8
t_{total}(s)	1666.6	850.9	439.5	231.5	123.5	67.8	36.8	21.5
S	1	1.96	3.79	7.20	13.5	24.6	45.2	77.4
ε (%)	—	97.9	94.8	90.0	84.3	76.8	70.7	60.5
ε_{vel} (%)	—	99.5	97.9	94.9	89.5	84.3	79.6	71.7
ε_p (%)	—	97.9	91.9	83.2	73.5	63.9	56.0	39.2
$\varepsilon_{k,\varepsilon}$ (%)	—	96.7	92.5	86.8	82.8	75.6	69.4	60.9
$\varepsilon_{scalars}$ (%)	—	97.1	94.3	88.9	84.7	77.2	70.5	61.5
ε_{prop} (%)	—	97.6	94.9	93.6	87.1	76.5	73.5	71.7

Table 2. Parallel performance of the code for the first 30 iterations using the MUSCL discretization scheme.

n_p	1	2	4	8	16	32	64	128
Partition	1x1x1	1x2x1	1x4x1	1x8x1	2x4x2	2x4x4	2x4x8	2x8x8
t_{total}(s)	1692.0	832.2	420.8	215.9	116.1	61.0	31.9	17.8
S	1	2.03	4.02	7.84	14.6	27.7	53.1	95.1
ε (%)	—	101.7	100.5	98.0	91.1	86.7	82.9	74.3
ε_{vel} (%)	—	97.7	97.8	97.1	91.3	88.7	86.7	82.1
ε_p (%)	—	95.3	84.1	70.7	59.2	49.9	41.9	28.6
$\varepsilon_{k,\varepsilon}$ (%)	—	103.0	102.2	100.0	93.2	89.4	86.0	78.5
$\varepsilon_{scalars}$ (%)	—	106.8	106.4	104.5	98.2	94.2	90.9	84.5
ε_{prop} (%)	—	97.9	95.7	93.6	86.8	80.6	76.9	71.8

the enthalpy, mixture fraction and mixture fraction variance equations; v) the calculation of the mean properties, namely the turbulent viscosity and the mean values of density, temperature and chemical species concentrations. The efficiencies of these five tasks are referred to as ε_{vel}, ε_p, $\varepsilon_{k,\varepsilon}$, $\varepsilon_{scalars}$ and ε_{prop}, respectively.

It can be seen that the efficiency of the pressure correction task is the lowest one, and decreases much faster than the efficiencies of the other tasks when the number of processors increases. The reason for this behaviour is that the amount of data to be communicated associated with this task is quite large, as discussed in [24-25]. Therefore, the corresponding efficiency is strongly affected by the number of processors. The computational load of this task is independent of the discretization scheme of the convective terms because the convective fluxes across the faces of the control volumes are determined by means of the interpolation procedure of Rhie and Chow [26] and there is no transport variable to be computed at those faces as in the other transport equations. However, the communication time is larger for the MUSCL scheme, due to the larger halo region. Therefore, the task efficiency is smaller for the MUSCL scheme, in contrast to the overall efficiency.

The efficiency of the properties calculation is slightly higher for the hybrid scheme if n_p=16, and equal or slightly lower in the other cases, but it does not differ much from one scheme to the other. This is a little more difficult to interpret since the computational load of this task is also independent of the discretization scheme and the communication time is larger for the MUSCL scheme. Hence, it would be expected a smaller efficiency in the case of the MUSCL scheme, exactly as observed for the pressure task. But the results do not confirm this expectation. It is believed that the reason for his behaviour is the following.

If the turbulent fluctuations are small, the mean values of the properties (e.g., density and temperature) are directly obtained from the mean mixture fraction, neglecting those fluctuations. If they are significant, typically when the mixture fraction variance exceeds 10^{-4}, then the mean values are obtained from interpolation of the data stored in tabular form. This data is obtained *a priori* accounting for the turbulent fluctuations for a range of mixture fraction and mixture fraction variance values. Although the interpolation of the stored data is relatively fast, it is still more time consuming than the determination of the properties in the case of negligible fluctuations. Therefore, when the number of grid nodes with significant turbulent fluctuations increases, the computational load increases too. Since the calculations start from a mixture fraction variance field uniform and equal to zero, a few iterations are needed to increase the mixture fraction variance values above the limit of 10^{-4}. The results given in tables 1 and 2 were obtained using a different number of iterations, 30 for the MUSCL scheme and 80 for the hybrid scheme. So, it is expected that in the former case the role of the turbulent fluctuations is still limited compared to the last case. This means that the computational load per iteration will be actually higher for the hybrid scheme, rather than identical in both cases as initially assumed. This would explain the similar task efficiency observed for the two schemes.

The three remaining tasks, i), iii) and iv), exhibit a similar behaviour, the efficiency being higher for the calculations using the MUSCL scheme. This is explained exactly by the same reasons given for the overall efficiency. In both cases, the efficiency of these tasks is higher than the overall efficiency, compensating the smaller efficiency of the pressure task. For a small number of processors the efficiency of these tasks slightly exceeds 100%. This has also been found by other researchers and is certainly due to a better use of cache memory.

Tables 3 and 4 summarize the results obtained for a complete run, i.e., for a converged solution, using 32, 64 and 128 processors. Convergence is faster if the hybrid scheme is employed, as expected. Regardless of the discretization scheme, there is a small influence of the number of processors on the convergence rate, and although this rate tends to decrease for a large number of processors, it does not change monotonically. The complex interaction between different phenomena and the non-linearity of the governing equations may be responsible for the non-monotonic behaviour which has also found in other studies. Since the smaller number of processors used in these calculations was 32, the efficiency and the speedup were

Table 3. Parallel performance of the code using the hybrid discretization scheme

n_p	32	64	128
Partition	2x4x4	2x4x8	2x8x8
n_{iter}	1758	1749	1808
t_{total} (s)	1982	1107	701
S_{rel}	1	1.79	2.83
ε_{rel} (%)	—	89.5	70.7
$\varepsilon_{rel,\ vel}$ (%)	—	95.0	83.0
$\varepsilon_{rel,\ p}$ (%)	—	86.3	58.9
$\varepsilon_{rel,\ k,\ \varepsilon}$ (%)	—	91.6	78.1
$\varepsilon_{rel,\ scalars}$ (%)	—	90.9	77.5
$\varepsilon_{rel,\ prop}$ (%)	—	99.1	92.6
$\varepsilon_{rel,\ radiation}$ (%)	—	82.7	58.1

Table 4. Parallel performance of the code using the MUSCL scheme

n_p	32	64	128
Partition	2x4x4	2x4x8	2x8x8
n_{iter}	3244	3215	3257
t_{total} (s)	7530	4119	2524
S_{rel}	1	1.83	2.98
ε_{rel} (%)	—	91.4	74.6
$\varepsilon_{rel,\ vel}$ (%)	—	98.1	90.7
$\varepsilon_{rel,\ p}$ (%)	—	82.8	54.5
$\varepsilon_{rel,\ k,\ \varepsilon}$ (%)	—	96.5	86.4
$\varepsilon_{rel,\ scalars}$ (%)	—	97.3	88.3
$\varepsilon_{rel,\ prop}$ (%)	—	96.1	87.9
$\varepsilon_{rel,\ radiation}$ (%)	—	75.6	49.0

computed taking the run with 32 processors as a reference. This means that the values presented in tables 3 and 4, denoted by the subscript rel, are relative efficiencies and speedups. The relative speedup is higher when the MUSCL scheme is employed, in agreement with the trend observed in tables 1 and 2 for the first few iterations.

There are two tasks that exhibit a much smaller efficiency than the others: the solution of the pressure correction equation and the calculation of the radiative heat transfer. The low efficiency of the radiative heat transfer calculations is due to the decrease of the convergence rate with the increase of the number of processors [19-20]. The radiation subroutine is called with a certain frequency, typically every 10 iterations of the main loop of the flow solver (SIMPLE algorithm). The radiative transfer equation is solved iteratively and a maximum number of iterations, 10 in the present work, is allowed. If the number of processors is small, convergence is achieved in a small number of iterations. But when the number of processors is large, the limit of

10 iterations is achieved, and a number of iterations smaller than this maximum is sufficient for convergence only when a quasi-converged solution has been obtained. Both the pressure and the radiation tasks have a lower partial efficiency if the MUSCL scheme is used. In fact, the computational effort of these tasks is independent of the discretization scheme, and the communication time is higher for the MUSCL scheme. The same is true, at least after the first few iterations, for the properties task. The other tasks (momentum, turbulent quantities and scalars) involve the solution of transport equations and their computational load strongly depends on how the convective terms are discretized. Hence, their efficiencies are higher than the overall efficiency, the highest efficiencies being achieved for the MUSCL scheme.

5 Conclusions

The combustion chamber of a power station boiler was simulated using a Cray T3D and a number of processors ranging from 1 to 128. The convective terms of the governing equations were discretized using either the hybrid central differences/upwind or the MUSCL schemes, and a comparison of the parallel efficiencies attained in both cases was presented. The MUSCL scheme is more computationally demanding, and requires more data to be exchanged among the processors, but it yields higher speedups than the hybrid scheme. An examination of the computational load of different tasks of the code shows that two of them are controlling the speedup. These are the solution of the pressure correction equation, which requires a lot of communications among processors, and the calculation of the radiative heat transfer, whose convergence rate is strongly dependent on the number of processors. The efficiency of these tasks, as well as the efficiency of the properties calculation task, is higher for the hybrid than for the MUSCL schemes. On the contrary, the efficiency of the tasks that involve the solution of transport equations is higher for the MUSCL than for the hybrid scheme, and it is also higher than the overall efficiency.

Acknowledgements

The code used in this work was developed within the framework of the project Esprit No. 8114 — HP-PIPES sponsored by the European Commission. The calculations performed in the Cray T3D at the University of Edinburgh were supported by the TRACS programme, coordinated by the Edinburgh Parallel Computing Centre and funded by the Training and Mobility of Researchers Programme of the European Union.

References

1. Coelho, P.J. and Carvalho, M.G.: Application of a Domain Decomposition Technique to the Mathematical Modelling of a Utility Boiler. International Journal for Numerical Methods in Engineering, 36 (1993) 3401-3419.
2. Leonard, B.P. and Drummond, J.E.: Why You Should Not Use "Hybrid" Power-law, or Related Exponential Schemes for Convective Modelling — There Are Much Better Alternatives. Int. J. Num. Meth. Fluids, 20 (1995) 421-442.
3. Harter, A., Engquist, B., Osher S., and Chakravarthy S. : Uniformly High Order Essentialy Non-Oscillatory Schemes, III. J. Comput Phys., 71 (1987) 231-303.
4. Gaskell, P.H. and Lau, A.K.C.: Curvature-compensated Convective Transport: SMART, a New Boundedness-transport Algorithm. Int. J. Num. Meth. Fluids, 8 (1988) 617-641.
5. Zhu, J.: On the Higher-order Bounded Discretization Schemes for Finite Volume Computations of Incompressible Flows. Computer Methods Appl. Mech. Engrg., 98 (1992) 345-360.
6. Darwish, M.S.: A New High-Resolution Scheme Based on the Normalized Variable Formulation. Numerical Heat Transfer, Part B, 24 (1993) 353-371.
7. Lien, F.S. and Leschziner, M.A.: Upstream Monotonic Interpolation for Scalar Transport with Application to Complex Turbulent Flows. Int. J. Num. Meth. Fluids, 19 (1994) 527-548.
8. Choi, S.K., Nam, H.Y., Cho, M.: A Comparison of Higher-Order Bounded Convection Schemes. Computer Methods Appl. Mech. Engrg., 121 (1995) 281-301.
9. Kobayashi, M.H., and Pereira, J.C.F.: A Comparison of Second Order Convection Discretization Schemes for Incompressible Fluid Flow. Communications in Numerical Methods in Engineering, 12 (1996) 395-411.
10. Blake, R., Carter, J., Coelho, P.J., Cokljat, D. and Novo, P.: Scalability and Efficiency in Parallel Calculation of a Turbulent Incompressible Fluid Flow in a Pipe. Proc. 2nd Int. Meeting on Vector and Parallel Processing (Systems and Applications), Porto, 25-25 June (1996).
11. Van Leer, B.: Towards the Ultimate Conservative Difference Scheme. V. A Second-Order Sequel to Godunov's Method. J. Comput. Physics, 32 (1979) 101-136.
12. Libby, P.A. and Williams, F.A.: Turbulent Reacting Flows. Springer-Verlag, Berlin (1980).
13. Fiveland, W.A.: Discrete-ordinates Solutions of the Radiative Transport Equation for Rectangular Enclosures. J. Heat Transfer, 106 (1984) 699-706.
14. Fiveland, W.A.: The Selection of Discrete Ordinate Quadrature Sets for Anistropic Scattering. HTD-160, ASME (1991) 89-96.
15. Coelho, P.J., Gonçalves, J.M., Carvalho, M.G. and Trivic, D.N.: Modelling of Radiative Heat Transfer in Enclosures with Obstacles. International Journal of Heat and Mass Transfer, 41 (1998) 745-756.

16. Patankar, S.V.: Numerical Heat Transfer and Fluid Flow. Hemisphere Publishing Corporation (1980).
17. Khosla, P.K. and Rubin, S.G.: A Diagonally Dominant Second-order Accurate Implicit Scheme. Computers & Fluids, 1 (1974) 207-209.
18. Darwish, M.S. and Moukalled, F.H.: Normalized Variable and Space Formulation Methodology for High-Resolution Schemes. Numerical Heat Transfer, Part B, 26 (1994) 79-96.
19. Coelho, P.J., Gonçalves, J. and Novo, P.: Parallelization of the Discrete Ordinates Method: Two Different Approaches In Palma, J., Dongarra, J. (eds.): Vector and Parallel Processing - VECPAR'96. Lecture Notes in Computer Science, 1215. Springer-Verlag (1997) 22-235.
20. Gonçalves, J. and Coelho, P.J.: Parallelization of the Discrete Ordinates Method. Numerical Heat Transfer, Part B: Fundamentals, 32 (1997) 151-173.
21. Cassiano, J., Heitor, M.V., Moreira, A.L.N. and Silva, T.F.: Temperature, Species and Heat Transfer Characteristics of a 250 MWe Utility Boiler. Combustion Science and Technology, 98 (1994) 199-215.
22. Carvalho, M.G., Coelho, P.J., Moreira, A.L.N., Silva, A.M.C. and Silva, T.F.: Comparison of Measurements and Predictions of Wall Heat Flux and Gas Composition in an Oil-fired Utility Boiler. 25th Symposium (Int.) on Combustion, The Combustion Institute (1994) 227-234.
23. Coelho, P.J. and Carvalho, M.G.: Evaluation of a Three-Dimensional Mathematical Model of a Power Station Boiler. ASME J. Engineering for Gas Turbines and Power, 118 (1996) 887-895.
24. Coelho, P.J.: Parallel Simulation of Flow, Combustion and Heat Transfer in a Power Station Boiler. 4th ECCOMAS Computational Fluid Dynamics Conference, Athens, Greece, 7-11 September (1998).
25. Coelho, P.J., Novo, P.A. and Carvalho, M.G.: Modelling of a Utility Boiler using Parallel Computing. 4th Int. Conference on Technologies and Combustion for a Clean Environment, 7-10 July (1997).
26. Rhie, C.M., and Chow, W.L.: Numerical Study of the Turbulent Flow past an Airfoil with Trailing Edge Separation. AIAA J., 21 (1983) 1525-1532.

Parallel Implementation of Edge-Based Finite Element Schemes for Compressible Flows on Unstructured Grids

Paulo Lyra[1], Ramiro Willmersdorf[1], Marcos Martins[2], and Álvaro Coutinho[2]

[1] Departamento de Engenharia Mecânica - UFPE
Recife/PE, Brasil
prmlyra@npd.ufpe.br rbw@demec.ufpe.br
[2] Departamento de Engenharia Civil - COPPE/UFRJ
Rio de Janeiro/RJ, Brasil
marcos@civil01.coc.ufrj.br alvaro@coc.ufrj.br

Abstract. Some aspects of the parallel/vector implementation of an adaptive edge-based high-resolution scheme, for the solution of compressible Euler equations on unstructured grids, on current shared memory supercomputers are presented. We address the use of an alternative data structure, known as superedge, which groups together several edges and which attempts to find a good balance between *floating point* (*flop*) and *indirect addressing* (i/a) operations. It is shown that the practical usefulness of the flow solver has been dramatically improved by efficient implementation on high performence computer configurations and also that switching from edge-based to a superedge data structured is not worthwhile for codes which already have suficiently high rate between (*flop*) and (i/a).

1 Introduction

In recent years, there has been a significant level of research into the application of unstructured mesh methods to the simulation of fluid dynamic problems. For unstructured triangular and tetrahedral meshes, major progress has been made in the areas of automatic mesh generation and flow solver accuracy [1, 2]. However, the storage of mesh connectivity information increases the use of computer memory and indirect addressing to retrive local information required for the flow solver algorithm. To reduce (i/a) and memory requirements, finite element schemes based on edge-based data structures have been introduced by Morgan et al. [3], inspired on the finite volume schemes (see Barth in [4]). The use of an edge-based data structure also enables a straightforward implementation of upwind-biased schemes in the context of finite element methods.

In this paper, an upwind biased high-resolution flux-split algorithm is used as the general approach for constructing high-resolution schemes. An adaptive edge-based Galerkin finite element formulation is used as the building block for the multidimension generalization of the essentially one-dimensional upwinding

concepts. The resultant flow solver is used for the solution of compressible Euler equations on unstructured grids. A simple explicit time integration is adopted to drive the solution towards a steady-state. For a detailed description on the flow solver algorithm and related issues see Lyra [2].

The number of repeated evaluation of right-hand sides (RHS) or residuals is quite large and time consuming with explicit upwind-like schemes. The use of efficient data structures, searching algorithms and implementations is fundamental. The main steps adopted for a parallel/vector implementation on the CRAY J90 of the flow code using either an edge-based or the alternative superedges data structure are described. The techniques used are simple and try to reduce investiment in man-hours when re-writing the code for the use of alternative data structures [5]. These issues will be discussed in detail. Finally, we will present a comparative performance study, on different computer plataforms, of edge-based and triangular superedges schemes for the solution of a typical two dimensional model problem of a supersonic flow past a circular cylinder.

2 Numerical Solution Algorithm

2.1 Edge-Based Finite Element Formulation

Assuming that the spatial 2-D domain Ω is discretized into an unstructured assembly of linear triangular elements, after employing the Galerkin finite element approximation, the resultant discrete formulation can be conveniently expressed as

$$\left[M\frac{dU}{dt}\right]_I = -\sum_{S=1}^{m_I} \mathcal{L}_{II_S} \overbrace{\frac{(F_I^j \mathcal{S}_{II_S}^j + F_{I_S}^j \mathcal{S}_{II_S}^j)}{2}}^{F_{II_S}^S} + \langle \sum_{f=1}^{2} D_f (4\overline{F}_I^n + 2\overline{F}_{J_f}^n + F_I^n - F_{J_f}^n) \rangle_I \ . \qquad (1)$$

where an edge-based data structure has been employed instead of the conventional finite element data structure which is based on the connectivity of the elements. This allows a direct implementation of different types of standard 1-D upwind or centered shock-capturing methods within an unstructured grid context [2, 3].

The extension of the one dimensional upwinding concepts to two-dimensional generic discretisations consists on the use of an edge-based Galerkin finite element formulation together with different strategies to build a local 1-D like "structured" stencil by means of an interpolation reconstruction technique [6]. It is well known that the use of this data structure has additional beneficial effects, in terms of both processing time and memory requirements [3]. These

effects will be of particular importance when the extension of these methods to the solution of large scale three dimensional problems is done.

In (1), m_I is the number of sides connected to node I, the second term on the right hand side denotes a correction required for the nodes I which lie on the boundary of the computational domain and

$$C_{IIs} = (C^1_{IIs}, C^2_{IIs}) \; ; \;\; \mathcal{L}_{IIs} = |C_{IIs}| \; ; \;\; \mathcal{S}^j_{IIs} = C^j_{IIs}/|C_{IIs}| \; . \qquad (2)$$

Here, C^j_{IIs} and D_I are coefficients which depends on the finite element trial functions. From the asymmetry of the edge weights, the numerical discretization scheme can be immediately observed to possess a conservation property, in the sense that the sum of the contributions made by any interior edge is zero [3]. Finally, M represents the finite element, consistent, mass matrix.

Practical algorithms for the Euler equations can be produced by evaluating a convenient numerical flux $\mathcal{F}^{\mathcal{S}}_{IIs}$ in the direction of the weighting coefficient $\mathcal{S}^j_{IIs}$ in the place of the actual flux $F^{\mathcal{S}}_{IIs}$. In this work the flux difference scheme proposed by Roe [7] is adopted as the lower-order stable formulation.

2.2 High-Resolution Scheme

The MUSCL scheme (*Monotonic Upstream-centered Schemes for Conservation Laws*), proposed by Van Leer [8], is used to build high-resolution schemes. The higher-order slope-limited MUSCL can be defined through the numerical flux

$$\mathcal{F}^{\mathcal{S}}_{IIs} = \frac{1}{2}\{(F^j(\widehat{U}_L)\mathcal{S}^j_{IIs} + F^j(\widehat{U}_R)\mathcal{S}^j_{IIs}) - |A^{\mathcal{S}}(\widehat{U}_L, \widehat{U}_R)|\,(\widehat{U}_R - \widehat{U}_L)\} \; . \qquad (3)$$

where a piecewise linear reconstruction, with the introduction of non-linear limiters, is used to compute the limited interface values $\widehat{U}_L$ and $\widehat{U}_R$.

The key point of this class of higher-order procedure is the extension of the support of the lower-order stencil and the guarantee of the monotonicity property of the lower-order scheme. For a detailed description on the different formulations, for a discussion on several possibilities for the construction of the extended stencil, the choice of the limiter functions and other related issues see Lyra [2] and Lyra et al. [6]. Several different limiter functions, which may be computed using an upwind stencil or a symmetric stencil, are available. Here, the primitive variables are chosen to be limited for economical reasons. Numerical evidence supports this choice, since no oscillation, or very little oscillation, is present in the solution. The limit function given in [9] was used in the analysis presented in this paper.

Equation (1) represents the time evolution of the unknown vector $U_I(t)$ at node I of the mesh. A practical solution algorithm is then produced by further discretizing the time dimension, with a simple explicit time stepping scheme. The consistent finite element "mass" matrix M is replaced by the standard lumped (diagonal) mass matrix M_L. This enables a truly explicit time integration and does not alter the final steady state solution, which is of primary concern here.

For the steady-state analysis studied, local time stepping was used to accelerate the convergence rate towards steady-state.

3 The Computational Implementation and Other Issues

3.1 Mesh Generation and Adaption

The unstructured triangulations adopted for the 2-D computations were generated with the advancing front technique. An adaptive mesh enrichment procedure for steady state solution was used to improve the accuracy of the inviscid computations analysed. The error estimates are based upon concepts from interpolation theory and are used to control automatically the adaptivity. Further details about the mesh generator, error analysis involved in the procedure and about the adaptive procedure itself can be found in Morgan et al. [3].

3.2 Data Structure

The standard finite element data structured consists of the physical coordinates simply listed by node numbers, a list of the connectivity of each element and a list of boundary edges connectivities. With this geometrical and topological data, the integral terms that appear on the finite element formulation of the problem can be calculated with a loop over the elements and a loop over the boundary edges with the contributions to the nodes being accumulated during the process.

As an alternative to the element-based data structure, we can represent an unstructured grid in terms of an edge-based data structured. The physical coordinates are simply listed by node numbers and a list of boundary edge connectivities is adopted, but now the topology inside the domain is characterized through the edges and their connectivities. A significant reduction in gather/scatter costs and memory requirements can be realized by going from an element-based to an edge-based data structure (see Luo et al. [10], Morgan et al. [3] and Martins et al. [11]), this being more pronunced in three dimensional simulations.

The use of edge-based data structure in CFD, as opposed to the element-based data structure, has the following well stablished advantages: a) better performance of the numerical procedures in terms of computational efficiency (the computational effort necessary to evaluate and assemble edge contributions to the nodes is significantly reduced when compared with the element counterpart and the indirect addressing is also reduced); b) allows straightfoward construction of different flow solver algorithms, by generalizing one-dimensional algorithms (from centered to upwinding approaches); c) easy to implement numerical schemes for both 2D and 3D applications, also due to the 1-D like data structure; d) satisfaction of discrete conservation property.

Löhner [5] pointed out that the expected reduction in the total CPU cost can be partially lost if the indirect addressing overhead accounts for the major part of the total CPU requirements. With the design criterion of operating as much as

possible on gathered data. Löhner presents several alternative data structures. According to Löhner the adoption of stars data structure leads to very small i/a reduction with the penalty of the necessity of major code rewriting and with possible drawbacks in terms of large bandwidths and cache-misses. The use of chains data structure can lead to good i/a reduction but would prevent implementation on vector processors.

Analysing the main characteristics of the proposed alternative data structures suggested by Löhner [5], the superedge alternative seems to represent the best compromise, being suitable for scalar or vector implementations. Martins et al. [11] have demonstrated that migrating from an edge-based data structured to a superedge data structured would lead to at least 20% gain in CPU usage for three dimensional potential flows. We will address the use of superedges for the explicit solution the Euler equations using high-resolution schemes.

3.3 Pre-Processing

The generated grids, either initial or refined, are provided in the conventional element-based data structure format. Thus, a pre-processing of the grid must be undertaken before it can be used with an edge-based flow analysis algorithm. After the preprocessor stage the element-based data structure can be discarded. The pre-processor stage consists basically on the following steps:

1. Build the arrays with the grid and boundary topology, which are lists of edges and boundary faces with their respective connectivities;
2. Using superedges, group edges as superedges and organize the remaining edges;
3. Compute and store the edges and boundary faces weighting coefficients [2];
4. Find and store the required information necessary for the use of the dummy nodes;
5. Employ a colouring algorithm [12] to group the edges, superedges and boundary faces in such a way that no repetition in the node numbering occurs amongst items of the same group;

Remark 1. The information required to describe an unstructured mesh is minimal when using an edge-based data structure. A hash table searching technique is used to extract the edges from the original data structure.

Remark 2. As the number of edges in a group increases the longer and more complicate loops have to be implemented. Therefore, as in the words of Löhner [5] "a balance has to be struck between efficiency and code simplicity, clarity and maintenance". A triangle superedge was adopted so that, besides the above reasons, very simple grouping algorithm can be devised, i.e. grouping the three sides of a triangle as a superedge in such a way that as much three-edge groups as possible are formed and then organize the remaining edges [11].

Remark 3. The three nodes of the triangle that contains the dummy node, and two shape functions evaluated at the dummy node for the interpolation step, are

kept in memory for each of the two dummy nodes that belong to each side. This procedure represents a memory overhead of ten times the total number of sides in a 2-D computation, but such reconstruction approach is very robust for high speed flow simulation and recommended in such flow regimes. The alternating digital tree [12] algorithm is adopted for the searching operations required. Other reconstruction techniques which incur in no or very litle memory overhead [2, 6] can be employed.

Remark 4. The colouring algorithm [12] is used to prevent recurrence inside the loops used in the flow solution algorithm, therefore allowing vector and parallel processing of these loops.

3.4 Parallel/Vector Implementation

The operations performed inside the loops over the edges and boundary faces, which take place in the flow solver edge-based algorithm are: gather information from the nodes of each edge; operate on this information; scatter the results back to the nodes of the edges and add them to the nodal quantities.

These typical loops are entirely vectorizable provided each group (colour) of superedges and/or edges, or boundary faces, is executed separately and an appropriate compiler directive (e.g. *!DIR$ IVDEP*, for CRAY supercomputers) instructing the compiler is inserted before the vectorizable inner loop. Essentially, these loops would be written like the one shown in Fig. 1. Clearly, some details habe been omitted for conciseness.

```
***     Loop over all colours
        do 20 iblock = 1, nblock
***         Compute first and last edges in this colour
                          :
***         Loop over all edges in this color
!DIR$ IVDEP
            do 10 is = isfirst, islast
                inode1 = iside(is,1)
                inode2 = iside(is,2)
***             Compute contributions a1 from side is
                              :
                avar(inode1) = avar(inode1) + a1
                avar(inode2) = avar(inode2) - a2
10          continue
20      continue
```

Fig. 1. Loop over edges

where **nblock** is the number of colours into which the mesh was dividided, **isfirst** and **islast** are the first and last edges for each colour, **iside** is the

array with the nodal connectivity of edges, **avar** is the variable to be updated with this loop and **a1** and **a2** are the local contribuitions to **avar** of the nodes of edge **is**. The computation of the local contributions, not shown in Fig. 1, can demand a large number of operations, up to more than one hundred FORTRAN statements in some routines used in the second order scheme.

To implement superedge loops, all loops such as the one shown in Fig. 1 were modified to first loop over all triangular superedges, and then loop over the remaining edges of the mesh. The results of these modifications are loops such as the one shown in Fig. 2.

It is important to observe that the calculations for the local contributions of the nodes, also not shown in Fig. 2, are identical to those in the loop over the edges. So these changes could be made somewhat mechanically, using a "cut and paste" facility of a modern text editor. This is not recommended, however, since this procedure is very labour intensive, error prone and would result in extremely long procedures that seriously compromise the readability and the longer term maintenance of the computer program.

To avoid this problem, we collected all repeated computations into separated subroutines. This is a classical technique to improve code modularity and reusability, which has not been widely used in vector processing because compilers normally cannot vectorize loops with call to arbitrary subroutines. To overcome this serious drawback, we used the *inline* facility (present in most modern compilers, even for sequential machines), that expands the source code of the called routine into the body of the calling program, before compilation. The compiler therefore sees no function call and can vectorize and paralellize the loops as usual. This procedure allows easy implementation of different alternative data structures with very little effort for coding changes, once the pre-processed data is available.

The inner loops were distributed to multiple processors using the *autotasking* facilities available in CRAY machines. In the cases where loops were not automatically tasked, even when the update of the variables in the loop was independent because of the colouring, we inserted a compiler directive (*!MIC\$ DO ALL AUTOSCOPE VECTOR*), similar to that adopted for vectorization, to force the tasking of the loop.

The parallel regions in the code should be as long as possible, so that parallel start-up costs are reduced. Apart from the size of the problem being analysed, a proper balance on the number of components (edges, superedges or boundary faces) in each colour is fundamental for an effective parallel speed up. Longer parallel regions can be obtained if we apply a domain decomposition and parallel directives applied at a higher level in the program. This has not been implemented yet because it would imply into a more complex rewriting of the code, since we would need colourings on two levels, special treatment for the borders between domains and some other issues.

```
***     Loop over all colours
        do 20 iblock = 1, nblock
***         Compute first and last superedges in this colour
                        :
***         Loop over all superedges in this color
!DIR$ IVDEP
!MIC$ DO ALL AUTOSCOPE VECTOR
            do 10 is = isfirst, islast, 3
                inode1 = iside(is,1)
                inode2 = iside(is+1,1)
                inode3 = iside(is+2,1)
***             Compute contribution a1 from edge is
                                :
***             Compute contribution a2 from edge is+1
                                :
***             Compute contribution a3 from edge is+2
                                :
                avar(inode1) = avar(inode1) + a1 - a3
                avar(inode2) = avar(inode2) + a2 - a1
                avar(inode3) = avar(inode3) + a2 - a2
10          continue
20      continue

**      Loop over all colours of remaining edges
        do 40 iblock = 1, nblockr
***         Compute first and last edges in this colour
                        :
***         Loop over all remaing edges in this color
!DIR$ IVDEP
!MIC$ DO ALL AUTOSCOPE VECTOR
            do 30 is = isfirst, islast
                inode1 = iside(is,1)
                inode2 = iside(is,2)
***             Compute contribution a1 from side is
                                :
                avar(inode1) = avar(inode1) + a1
                avar(inode2) = avar(inode2) - a2
30          continue
40      continue
```

Fig. 2. Loop over superedges

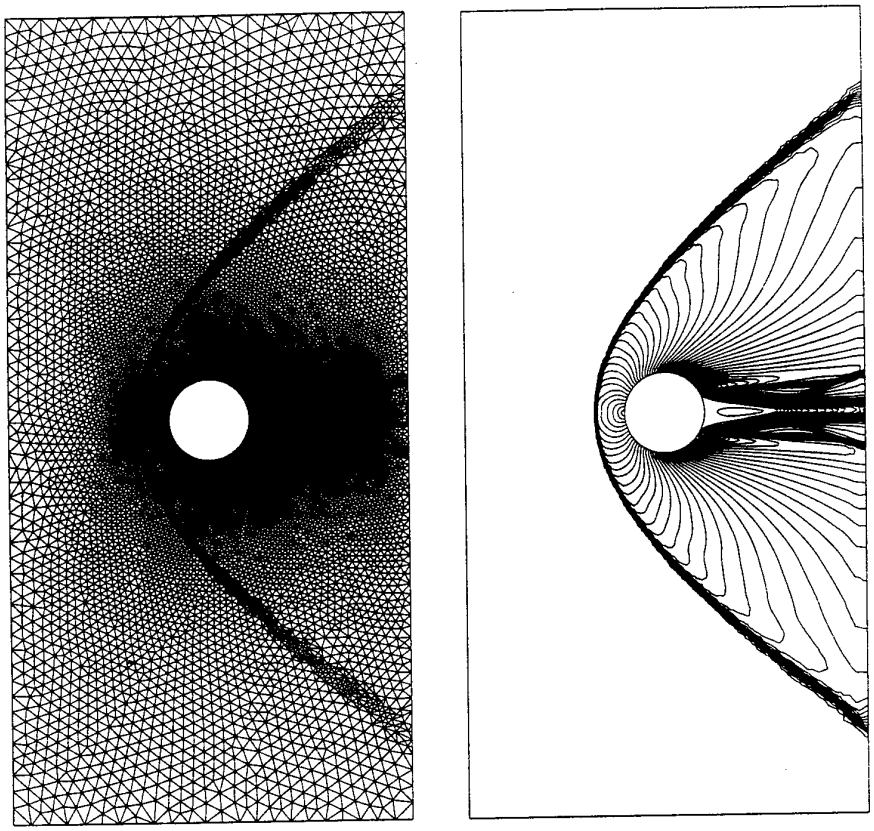


Fig. 3. Steady flow past a cylinder at a Mach number 3. Final mesh and corresponding Mach number contours.

4 Numerical Results and Conclusions

4.1 Numerical Application

This problem consists of a steady flow past a circular cylinder, at a free stream Mach number of 3. The presence of sonic, stagnation and rarefaction zones makes this problem challenging in terms of stability behavior. The final mesh, following one adaptation, together with the corresponding Mach number contours are shown in Fig. 3. This mesh consists of 24,979 elements and 12.651 nodes. Note that both the bow shock and the quasi-rarefaction zone behind the cylinder are well represented, with the recirculation and the weak shocks captured. A detailed discussion on the numerical prediction using different flow solvers can be found in [2] and is not of interest here.

4.2 Performance Studies

The CPU time for preprocessing the data, i.e., to form the edge or superedge data structure from the conventional element-based data structure and to do several pre-computations, is less then 0.5 % of the total CPU time for the analysis and so it is neglegible. The small time spent in the preprocessing step reflects the fact that it is done only once per analysis and also the use of data structures and algorithms which enable efficient sorting and searching operations (hash tables, alternating direct trees, etc.).

The mesh shown in the figure 3 has 37,630 edges. For the analysis using the superedge program the pre-processor was able to transform 86.75% of the edges into superedges (10881 triangle superedges), while 13.25% (4987) remained as single edges.

The performance of this test case was measured on three different computers, a 486 PC, a SUN UltraSparc I workstation and a CRAY J90 vector/parallel computer. The processing (CPU) times spent to simulate 5000 time steps are show in table 1. The analysis includes the program using either an edge or superedge data structure, considering both first-order and higher-order schemes.

Table 1. CPU times in seconds

	Edges		Superedges	
Computer	1^{st} order	2^{nd} order	1^{st} order	2^{nd} order
PC	34144	66837	32784	64190
SUN	4449	5272	3356	5167
Cray	2170	3543	2152	3496

By analysing table 1, we see that we can achieve only a minor advantage when using superedge instead of edge data structure for this application. The time savings for the PC is just around 4%, for the SUN workstation it is from 2% to 4% and for the single processor CRAY supercomputer it is approximatly 1%.

The ratio between the number of floating point and indirect addressing operations is already big enough for upwind-based schemes, as already expected. For instance, for the subroutine which computes Roe's approximate Riemann solver we have a ratio of approximately 6 and 8 floating points per indirect addressing for the edge and superedge data structure respectively.

We have analysed the first-order scheme for 1000 iterations on a single processor of CRAY J90 preventing vectorization. In this case we measured 2124 seconds and 1891 seconds for the CPU time with the edge and superedge data structure respectively. The difference was surprisingly much bigger then previously, being about 12% less CPU time with superedges. This is probably due to a bigger CPU overhead on indirect addressing when scalar processing is used, but CRAY's vector computers are known not to perform well without vectorization.

Regarding the vector performance, the Mflop/s (Million of floating point operations per second) rates were measured on single CPU computations employing CRAY's *perfview* tool. The sustained job performance on the CRAY J90 was around 57 Mflop/s for higher-order analysis and around 50 Mflop/s for first-order analysis. The use of either edge or superedge data structures had no significant impact on the Mflop rates. The code has sequential regions which justify the above values, somewhat lower than expected. The routines which have more intensive computations and which are highly vectorized reach up to 83 Mflop/s.

Parallel and theoretical speed-ups were estimated with CRAY's *atexpert* tool on four processor, using the F90 CRAY's compiler. The actual and theoretical speed-ups are presented in table 2. The theoretical speed-up in general is in good agrement with the speed-up obtained on a dedicated run.

Table 2. Parallel performance on Cray J90

	Edges		Superedges	
	1^{st} order	2^{nd} order	1^{st} order	2^{nd} order
Achieved Speed-up	1.9	2.3	1.44	1.5
Theoretical Speed-up	2.0	2.5	2.4	3.1
Serial Code Portion	33.9%	20.7%	21.6%	9.4%

It was observed that more then 99% of the serial portion of the code refers to the I/O routines. Such routines are called at each iteration or periodically for printing and flushing the residuals for history of convergence and the solution for possible re-starts of the analysis, respectively. Those subroutines could be easily changed, reducing substantially the serial portion of the code and improving vector and parallel performance. This was not done, however, to keep the code robust for new challenging applications.

The importance of load balancing in the efficiency of the program can be easily verified in the routine which solves the Riemann problem, using superedges. It has two main loops: one over the superegdes and another one over the remaining edges, as illustrated in Fig. 2. The second loop, over the edges, is exactly the same as the loop used to solve the Riemann problem in the edge based program. This loop parallelizes very well, and in the edge based program it reaches a speed-up of about 3.9. In the superedge program, however, the speed-up of the corresponding loop was only 1.9.

With superedges, the 4987 remaining edges were blocked into six unbalanced colours (2191; 1702; 678; 364; 43 and 9 edges each), an artifact of the colouring algorithm used; while with the edge program, which used a different colouring algorithm, the whole 37630 edges were blocked into ten colours with a balanced 3763 edges each. The increase on the number of processor cannot be efficient if the extra processors are assigned to loops so short that the overhead for setting up the parallel loops is significant. This shows the importance of balancing the

number of components for each colour and also that only for complex problems which demand very large meshes can we expect good scalabity and efficiency.

4.3 Conclusions

In this, paper we have addressed several important issues for simple and effective implementation of numerical schemes on current shared memory supercomputers. Superedges, and other alternative data structures, are interesting when the indirect addressing operations account for an important portion of the total computational cost.

The use of superedges when solving the Euler equations using explicit high-resolution upwind-based schemes, however, does not pay off. On a three dimensional generalization of the current formulation we might expect a slightly larger difference in the run times between edge and superedge data structure. Finally, when an implicit implementation of the flow solver algorithm is devised, using nonsymetrical solver such as GMRES, the matrix vector multiplication loop will present a smaller ratio between floating point and indirect addressing operations, and reducing i/a through the use of superedges might be worthwhile and could be attempted.

4.4 Acknowledgments

The authors would like to thank the support of FINEP, Financiadora de Estudos e Projetos, and FACEPE, Fundação de Amparo à Pesquisa do Estado de Pernambuco, the use of computing facilities of NACAD/UFRJ, Núcleo de Atendimento em Computação de Alto Desempenho, and the help of Mr. Paulo Tibério Bulhões of Silicon Graphics/Cray Research, Brasil.

References

[1] AGARD. Special Course on Unstructured Grid Methods for Advection Dominated Flows. Technical Report 787, France, 1992.

[2] P.R.M. Lyra. *Unstructured Grid Adaptive Algorithms for Fluid Dynamics and Heat Conduction*. PhD thesis, University of Wales - Swansea, 1994.

[3] K. Morgan, J. Peraire, and J. Peiró. Unstructured Grid Methods for Compressible Flows. In *Report 787 - Special Course on Unstructured Grid Methods for Advection Dominated Flows*. AGARD, 1992.

[4] T.J. Barth. Aspects of Unstructured Grids and Finite-Volume Solvers for the Euler and Navier-Stokes Equations. In *AGARD Report 787 on Special Course on Unstructured Grid Methods for Advection Dominated Flows*, pages 6.1–6.61, 1992.

[5] R. Löhner. Edges, Stars, Superedges and Chains. *Comp. Meth. Appl. Mech. Eng.*, 111:255–63, 1994.

[6] P.R.M. Lyra, O. Hassan, and K. Morgan. Unstructured Grid Adaptive Solutions of Hypersonic Viscous Flows. In *4th International Conference on Numerical Methods for Fluid Dynamics*, Oxford/UK, 1995.

[7] P.L. Roe. Approximate Riemann Solvers, Parameter Vectors and Difference Schemes. *J. Comp. Phys.*, 43:357–372, 1981.

[8] B. Van Leer. Towards the Ultimate Conservative Difference Scheme. V. A Second Order Sequel to Godunov's Method. *J. Comp. Phys.*, 32:101–136, 1979.

[9] J.L. Thomas. An Implicit Multigrid Scheme for Hypersonic Strong–Interaction Flowfields. In *Proc. of the Fifth Copper Mountain Conference on Multigrid Methods*, 1991.

[10] H. Luo, J.D. Baum, R. Löhner, and J. Cabello. Adaptive Edge-Based Finite Element Schemes for the Euler and Navier-Stokes Equations on Unstructured Grids. Technical Report 93-0336, AIAA, 1993.

[11] M.A.D. Martins, A.L.G.A. Coutinho, and J.L.D. Alves. Parallel Iterative Solution of Finite Element Systems of Equations Employing Edge-Based Data Structures. In *Proc. of the 8th SIAM Conf. on Parallel Processing for Scientific Computing*, 1997.

[12] J. Peiró, J. Peraire, and K. Morgan. FELISA SYSTEM:Reference Manual part1 – Basic Theory. Technical report, University of Wales Swansea Report CR/821/94, 1994.

Parallel 3D air flow simulation on workstation cluster

Jean-Baptiste Vicaire[1], Loic Prylli[1], Georges Perrot[1], and Bernard Tourancheau[2]*

[1] LHPC & INRIA REMAP, laboratoire LIP, ENS-Lyon 699364 Lyon - France
Loic.Prylli@ens-lyon.fr,
WWW home page: http://www.ens-lyon.fr/LIP
[2] LHPC & INRIA REMAP, laboratoire LIGIM bat710, UCB-Lyon
69622 Villeurbanne - France
Bernard.Tourancheau@inria.fr,
WWW home page: http://lhpca.univ-lyon1.fr

Abstract. *Thesee* is a 3D panel method code, which calculates the characteristic of a wing in an inviscid, incompressible, irrotational, and steady airflow, in order to design new paragliders and sails.
In this paper, we present the parallelization of *Thesee* for low cost workstation/PC clusters. *Thesee* has been parallelized using the ScaLAPACK library routines in a systematic manner that lead to a low cost development. The code written in C is thus very portable since it uses only high level libraries. This design was very efficient in term of manpower and gave good performance results. The code performances were measured on 3 clusters of computers connected by different LANs : an Ethernet LAN of SUN SPARCstation, an ATM LAN of SUN SPARCstation and a Myrinet LAN of PCs. The last one was the less expensive and gave the best timing results and super-linear speedup.

1 Introduction

The aim of this work is to compare the performance of various parallel platforms on a public domain aeronautical engineering simulation software similar to those routinely used in the aeronautical industry where the same numerical solver is used, with a less user-friendly interface, which results in a more portable code (smaller size, no graphic library).

Parallel *Thesee* is written in C with the ScaLAPACK[3] library routines and can be run on top of MPI[14] or PVM[6], thus the application is portable on a wide range of distributed memory platforms.

We introduce in the following the libraries that are necessary to understand the ScaLAPACK package. Then we give an insight of the parallelization of the code. Tests are presented before the conclusion.

* This work was supported by EUREKA contract EUROTOPS, LHPC (Matra MSI, CNRS, ENS-Lyon, INRIA, Région Rhône-Alpes), INRIA Rhône-Alpes project REMAP, CNRS PICS program, CEE KIT contract

2 Software libraries

2.1 LAPACK, BLAS and BLACS libraries

The BLAS (Basic Linear Algebra Subprograms) are high quality "building block" routines for performing basic vector and matrix operations. Level 1 BLAS do vector-vector operations, Level 2 BLAS do matrix-vector operations, and Level 3 BLAS do matrix-matrix operations.

LAPACK[1] provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) are also provided. The LAPACK implementation used as much as possible the BLAS building block to ensure efficiency, reliability and portability.

The BLACS (Basic Linear Algebra Communication Subprograms) are dedicated to communication operations used for the parallelization of the level 3 BLAS or the ScaLAPACK libraries. They can of course be used for other applications that need matrix communication inside a network. BLACS are not a multi-usage library for every parallel application but an efficient library for matrix computation.

In the BLACS, processes are grouped in one or two dimension grids. BLACS provide point to point synchronous receive, broadcast and combine. There is also routines to build, modify or to consult a grid. Processes can be enclosed in multiple overlapping or disjoint grids, each one identified by a context. Different release of BLACS are available on top of PVM,MPI and others. In this project, BLACS are used on top of PVM or MPI.

2.2 The ScaLAPACK library

ScaLAPACK is a library of high-performance linear algebra routines for distributed memory message passing MIMD computers and networks of workstations supporting PVM and/or MPI . It is a continuation of the LAPACK project, which designed and produced analogous software for workstations, vector supercomputers, and shared-memory parallel computers. Both libraries contain routines for solving systems of linear equations, least squares problems, and eigenvalue problems. The goals of both projects are efficiency (to run as fast as possible), scalability (as the problem size and number of processors grow), reliability (including error bounds), portability (across all important parallel machines), flexibility (so users can construct new routines from well-designed parts), and ease of use (by making the interface to LAPACK and ScaLAPACK look as similar as possible). Many of these goals, particularly portability, are aided by developing and promoting standards , especially for low-level communication and computation routines. LAPACK will run on any machine where the BLAS are available, and ScaLAPACK will run on any machine where both the BLAS and the BLACS are available.

3 Implementation

The *Thesee* sequential code [8] uses the 3D panel method (also called the singularity element method), which is a fast method to calculate a 3D low speed airflow[9]. It can be separated in 3 parts :

Part P1 The fill in of the element influence matrix from the 3D mesh. Its complexity is $O(n^2)$ with n the mesh size (number of nodes). Each matrix element gives the contribution of a double layer (source + vortex) singularity distribution on facet i at the center of facet j.

Part P2 The LU decomposition of the element influence matrix and the resolution of the associated linear system $(O(n^3))$, in order to calculate the strength of each element singularity distribution.

Part P3 The speed field computation. Its complexity is $(O(n^2))$, because the contribution of every nodes has to be taken into account for the speed calculation at each node. Pressure is then obtained using the Bernoulli equation

Each of these parts are parallelized independently and are linked together by the redistribution of the matrix data. For each part, the data distribution is chosen to insure the best possible efficiency of the parallel computation.

The rest of the computation, is the acquisition of the initial data and the presentation of the results. Software tools are uses to built the initial wing shape and to modify it as the results of the simulation gives insight that are valuable. The results are presented using a classical viewer showing the pressure field with different colors and with a display of the raw data in a window.

3.1 Fill in of the influence matrix

The 3D meshing of the wing is defined by an airfoils file, i.e. by points around the section of the wing that is parallel to the airflow, in order to define the shape of the wing in this section, as in the NACA tables[1]. Thus two airfoils delimit one strip of the wing, i.e. the narrow surface between the airfoils. The wing is then divided in strips and each strip is divided in facets by perpendicular divisions joining two similar points of the airfoils. This meshing is presented in Figure 1. Notice that the points are not regularly distributed in order to have more facets, thus more precision in the computation, in areas where the pressure gradient is greater.

[1] http://www.larc.nasa.gov/naca/

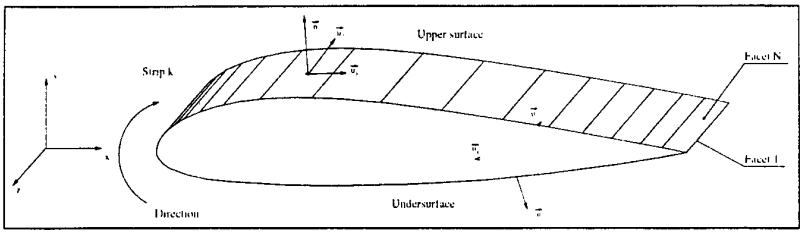


Fig. 1. Strip of the meshing

During the computation, for each facet the whole wing must be examined (every other facet). Since the wing is symmetrical, the computation is made with a half-wing, thus as the dimension of the equation system is $2 \times n$, its size is divided by four. The results for the whole wing are then deduced from those of the half-wing. Let K be the number of strip, and N be the number of facets per strip. The sequential code for the fill in of the influence matrix is made of nested loops :

```
/* computation of the influence coefficient (source)*/
 for (i1=1;i1<=K/2;i1++) {     /*for each strip */
    cste1=(i1-1)*(N+2)+1;
    for (j1=cste1;j1<=(cste1+N-1);j1++) {
      /* 2 loops to examine each point of the half wing */
      for (i2=1;i2<=K/2;i2++) {
         cste2=(i2-1)*(N+2)+1;
         for(j2=cste2;j2<=(cste2+N);j2++)
         {
           ..........................
           M[ind++]=.....................
         }
```

Fig. 2. Sequential code for the influence matrix fill in.

The computation of the influence coefficient $M[i]$ for one facet is independent of the other coefficients. The external loop can be split up straightforwardly in order to parallelize the computation on different processors. Each processor will then compute the facets of a given number of strips. The strips are assigned to a processor according to their number. For instance, with 4 processors and 20 strips to compute, each processor will compute five strips; n°1 strip from 1 to 5, n°2 strip from 6 to 10 etc...

Proc n°	from strip n°	to
1	1	$\lceil \frac{K/2}{Nbproc} \rceil$
2	$\lceil \frac{K/2}{Nbproc} \rceil + 1$	$\lceil 2 * \frac{K/2}{Nbproc} \rceil$
...	...	...
n	$\lceil (n-1) * \frac{K/2}{Nbproc} \rceil + 1$	$\lceil n * \frac{K/2}{Nbproc} \rceil$

Fig. 3. Simple assignation of the strips to compute to the processors.

Hence the external loop becomes:

```
for (i1=ceil((MyPRow)*(K/2)/((NPRow)))+1;
     i1<=ceil((MyPRow+1)*(K/2)/(NPRow));
     i1 ++)
```

Fig. 4. The parallel version of the external loop corresponding to Figure 3 assignment.

The data needed for these computations (the initial meshing of the wing) are distributed to each processor using the PDGEMR2D routine from the ScaLAPACK parallel library [3], the results is then gathered with the same routine. This routine provides the distribution of data between virtual grids of processors with any kind of block cyclic data distribution. We used it from the initial "grid" of size 1×1 witch contains the matrix to compute, to the computation virtual grid of processors with 2 to 4 processors arranged in a 1×2 or 1×4 shape with, for instance, a full block data distribution. Notice that this data distribution introduces no other communication cost in this part because it is embarrassingly parallel.

The second inner step of this part, the computation of doublet influence coefficient, is realized with the same method, splitting the outer loop and using the PDGEMR2D routine for the data repartition.

3.2 Influence matrix resolution

This part mainly consists in the resolution of a linear algebra system (Part P2). In the sequential version of *Thesee* it is carried out with a simple call of the *DGESV* routine from LAPACK [1]. DGESV solves a linear equation system with a LU factorization and then a back-solve substitution.

```
DGESV_(&n, &nrhs, &M[1], &lda, &ipiv[1], &B[1], &ldb, &info);
```

In this procedure DGESV solves the $M * X = B$ system of equations and stores the results in the `B` vector.

The parallel solve routine *PDGESV* from ScaLAPACK provides the same system resolution with a parallel LU decomposition on block cyclically distributed

data on a virtual grid of processors. A redistribution of the data on each processor is then needed to use the parallelized resolution.

The main idea is that each processor involved in the resolution holds a submatrix of the matrix `M` to solve. The processors of the parallel machine with P processors are presented to the user as a linear array of process IDs, labeled 0 through (P-1). It is often more convenient while doing matrix computations to map this 1-D array of P processes into a logical two dimensional process mesh, or grid while doing matrix computation. This grid will have R processor rows and C processor columns, where $R * C = G <= P$. A processor can now be referenced by its coordinates within the grid (indicated by the notation i, j, where $0 <= i < R$, and $0 <= j < C$). An example of such a mapping is shown in Figure 5.

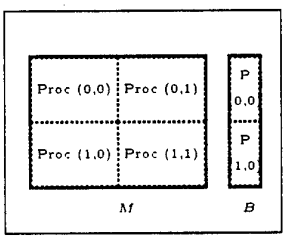


Fig. 5. Example of redistribution with a 2×2 grid

A processor can be a member of several overlapping or disjoint virtual grids during the computation, each one identified by a *context*.

The ScaLAPACK library uses a block cyclic data distribution on a virtual grid of processors in order to reach a good load-balance, good computation efficiency on arrays, and an equal memory usage between processors. The load-balance is insure by the cyclic distribution that gives to each processor matrix elements that are comming from "different" locations of the matrix (compared to a classical full block decomposition). The communication efficiency is obtained because in a cyclic distribution, the row and column shape of the matrix is preserved, so most of communication of 1D arrays can happen without a complex index computation (see [4, 5, 13, 2] among others). We ran tests in order to choose the best grid shape and the best block size of the data distribution for our problem on each of the platform.

Matrices and arrays are then wrapped by blocks in all dimensions corresponding to the processor grid using the PDGEMR2D routine. Figure 3.2 illustrates the organization of the block cyclic distribution of a 2D array on a 2D grid of processors.

In the parallel version of . we used the following parameters for the data distribution of the LU factorization: the block size is 32×32 and the processor

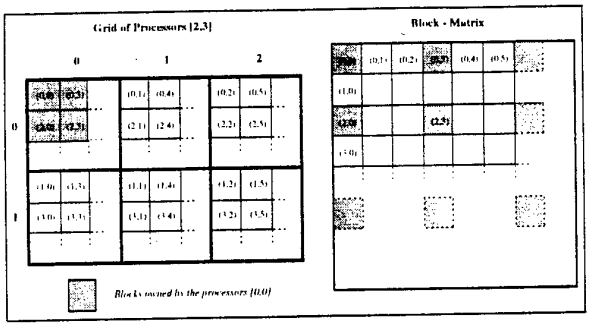


Fig. 6. The block cyclic data distribution of a 2D array on a 2×3 grid of processors.

grid shape is a 1D-grid that gave the best overall computation timings (for further information see[10]).

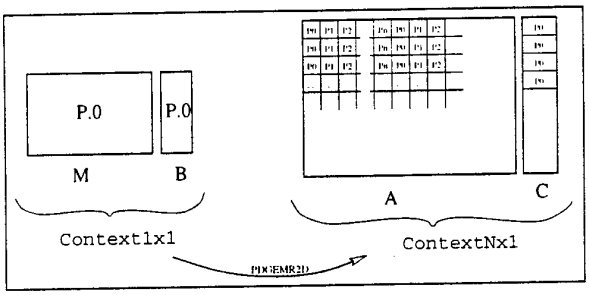


Fig. 7. Data redistribution inside parallel Thesee

In the first parts of parallel , the global system matrix (`M`) is hold in 1×1 grid by the proc 0 (`Context1x1`). Then, it is distributed over a $1 \times NbProc$ grid (`Context1xN`) with the *PDGEMR2D*[2]. The same operation is also realized for the `B` vector. Then each processor call the PDGESV routine from ScaLAPACK instead of DGESV from LAPACK. After this, the solution vector is distributed on the local sub-matrix of `B`. A new call to PDGEMR2D is needed to gather the solution sub-vectors from the `Context1xN` grid to the `Context1x1` grid.

3.3 Speed computation

The speed computation procedure (Part P3) is made of loops to compute the speed array (`S`) and the potential array (`Pot`).

The sequential code uses two nested loops for the speed array computation:

[2] Parallel Double GEneral Matrix Redistribution (from ScaLAPACK)

```
for(i2=1 ; i2<=K/2 ; i2++) {
   for(j2=1 ; j2<=N+1 ; j2++) {
      ........
      S[(i2-1)*(N+1)+j2]=...;
   }
}
```

Fig. 8. Sequential code of the speed computation

The computation of an element of the speed array is independent from all the others, giving us another nice embarrassingly parallel problem. The external loop is thus split like in Part P1 and each processor is assigned a given number of strips to compute. The computation code of the speed array is modified as shown in Figure 9.

```
/* number of  the first strip computed by the proc */
 ideb=ceil((MyPRow)*(K/2)/((NPRow)))+1;

 /* speed computation loop */
 for(i2=ceil((MyPRow)*(K/2)/((NPRow)))+1 ;
    i2<=ceil((MyPRow+1)*(K/2)/(NPRow));
    i2++)
  {
    for(j2=1 ; j2<=N+1 ; j2++)
      {
       .........
       S[(i2-ideb)*(N+1)+j2]=.......
      }
  }
```

Fig. 9. Parallelization of the external loop of the Figure 8

The computation of the potential array is done with the same method.

4 Performances tests

The tests were run on 2 different platforms with 3 different networks: an Ethernet and ATM network of SUN Sparc 5 85MHz with Solaris and an Ethernet and Myrinet network of Pentium-Pro 200MHz running Linux. The lower level computation libraries BLAS were an optimized version on the SUNs and a compiled version on the Pentium-Pros.

The efficiency of the fill in of the influence matrix and of computation of the speed and potential arrays are roughly the same on every configuration. The code for these parts is "embarrassingly parallel" and thus the speed-up is almost equal

to the number of processor. Whereas the LU factorization which involves a lot of communications is highly dependent of the network software and hardware.

4.1 Optimal block size

There is only slight differences between the different block sizes performances on such small platforms with our problem sizes. However a block size of 32×32 was the optimal on every configuration (i.e. Ethernet, ATM, Myrinet).

We present in Table 10 the timing for the LU resolution with a system 1722×1722 which correspond to our production problem size.

	Timings		
Block size	$IP/Ethernet^1$	IP/ATM^1	$IP-BIP/Myrinet^2$
8×8	71.3	67.7	30.1
16×16	68.9	61.3	29.5
32×32	68.3	61.1	28.1
64×64	74.4	66.8	34.1

Fig. 10. Timings of the LU resolution for different block and problem sizes.

The results show the advantage of the new Pentium-Pro200 generation over the rather old Sparc85 and gives the better block size for each configuration.

4.2 Comparison between Ethernet and ATM

We present here the timing results of the whole computation using the different platforms. First, we compare the networks with the same processor kind (SUN Sparc 5) over PVM.

The gain obtained with ATM is small because the startup time of this two network is similar to the one on Ethernet. This is the big part of the communication delay. However, the speedup obtained is not neglectable while using the code in production because the response time is critical.

4.3 Comparison between Ethernet and Myrinet

We present here the timings obtained on the PC platform. The Myrinet network is driven by the BIP[11, 12] (Basic Interface for Parallelism) software. BIP is a

[1] on SUN Sparc
[2] on Pentium Pro
[3] sequential version

System size	Timings IP/Ethernet (s)	IP/ATM (s)
4 Proc 902 × 902	16.3	13.5
2 Proc 902 × 902	18.8	16.2
1 Proc 902 × 902	20.3[3]	20.3[3]
4 Proc 1722 × 1722	68.3	61.1
2 Proc 1722 × 1722	108.9	95.9
1 Proc 1722 × 1722	131.8[3]	131.8[3]

Fig. 11. Timings of the whole computation.

System size	Speedup IP/Ethernet	IP/ATM
4 Proc 902 × 902	1.24	1.50
2 Proc 902 × 902	1.07	1.25
4 Proc 1722 × 1722	1.92	2.15
2 Proc 1722 × 1722	1.21	1.37

Fig. 12. Speedup of the whole computation.

new protocol that provides a small parallel API implemented on the Myrinet network. Other protocol layers are implemented for the classical interfaces. BIP delivers to the application the maximal performance achievable by the hardware using a low latency zero copy mechanism. An IP-BIP stack has been build on top of BIP. As well, a port of MPI-CH was realized with MPI-BIP[15, 12, 7]. The results of parallel þhave been measured with PVM over IP/Ethernet and PVM over IP-BIP/Myrinet and MPI over BIP/Myrinet. This gives an idea of the portability of our code that uses library calls that are available in IP, PVM, MPI, ...

System size	Timings IP/Ethernet (s)	IP-BIP/Myrinet (s)	MPI-BIP/Myrinet (s)
4 Proc 902 × 902	10.2	4.7	3.1
2 Proc 902 × 902	9.6	6.7	5.0
1 Proc 902 × 902	10.0[3]	10.0 [3]	10.0[3]
4 Proc 1722 × 1722	45.9	28.1	21.3
2 Proc 1722 × 1722	56.4	44.3	38.1
1 Proc 1722 × 1722	87.4[3]	87.4 [3]	87.4[3]

Fig. 13. Timings with Myrinet.

System size	Speedup IP/Ethernet	BIP-IP/Myrinet	MPI-BIP/Myrinet
4 Proc 902 × 902	0.97	2.10	3.24
2 Proc 902 × 902	1.03	1.49	1.97
4 Proc 1722 × 1722	1.90	3.10	4.09
2 Proc 1722 × 1722	1.54	1.97	2.29

Fig. 14. Speedup with Myrinet.

First notice the advantage of the Pentium-Pro speed against the Sparc on sequential numbers.

Not surprisingly the best results are achieved with MPI-BIP that provides a very low $9\mu s$ latency for the basic send communication. The gain on large problem size and platform is more than 50% over the Ethernet run, leading to a super-linear speed-up.

This can be explained by a better cache hit ratio in the parallel version of the code. As the matrix is distributed cyclically on the processors, the computation occurs on blocked data that fits better in the cache during the LU decomposition, leading to a better use of the processor's pipeline units. Moreover, an overlap of the communications is done in the parallel LU decomposition, this overlap can be (relatively) increased because the total amount of the communication cost is greatly reduced with the Myrinet+BIP platform.

These outstanding results show that low-level access to high-speed network is essential to achieve the best possible performances while doing parallel computation.

4.4 Industrial use of the code

We ran the code on an industrial version of the PC-Myrinet cluster, the POPC (Pile of PCs) machine designed by Matra. This architecture is a little bit slower than the original test-bed but gives interesting results about the scalability of the code. When using the compiled BLAS kernels, depending of the data size of the problem, there is little interest in going further than 6 processors. When using an optimized version of the BLAS designed for the Pentium processors, the timings dropped down by a factor of more than 2 for the small wing and more than 3 for the big one and there is little gain going for more than 4 processors with the small wing and more than 8 processors for the big one. From a user point of view, the elapse time was be decreased by a factor of 25 (from more than a minute to 5 second). This almost an immediate answer will drastically improve the production iterative process of the new wing shapes for the cost of a few PCs, the use of a good BLAS kernel, and a very simple parallelization method.

These outstanding results show that low-level access to high-speed network is essential to achieve the best possible performances while doing parallel computation.

System size	Number of processors					
	1	2	4	6	8	10
wing 902	8.97	5.11	3.11	2.36	2.12	1.79
wing 902 (optimized BLAS)	4.199	2.71	1.88	1.59	1.39	1.31
wing 1722	75.72	38.17	21.07	14.11	12.63	
wing 1722 (optimized BLAS)	25.337	14.6	8.09	6.32	5.41	5.10

Fig. 15. Timings of the whole execution on the POPC Pile of PC machine

5 Conclusion

We described our work on the parallelization of an air flow 3D simulation that use the singularity method that is well suited for low speed airflows.

We presented a very easy and clean way to parallelized such numerical code, using only parallel library routines (this requires the sequential code to be written with sequential library routines too), loop splitting and calls to a data redistribution routine. The parallel code is thus portable (we ran it on 3 different . IP. PVM. MPI) and efficient (super-linear speedup over Myrinet).

We demonstrate that low cost parallel hardware and good software can lead to significant improvement for production codes, starting from a more than 2 minutes delay and going to 21s on a four PCs platform with Myrinet.

Our future work will consist in the automatization of this parallelization process of numerical code using library routines with a software tool.

References

1. Anderson, Bai, Bischof, Demmel, Dongarra, Du Croz, Greenbaum, Hammarling, McKenney, Ostrouchov, and Sorensen. *LAPACK Users' Guide*. SIAM, 1994. http://www.netlib.org/lapack/.
2. E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. Van de geijn. LAPACK for distributed memory architecture. In *Fifth SIAM Conference on Parallel Processing for Scientific Computing*. USA, 1991.
3. Blackford, Choi, Cleary, d'Azevedo, Demmel, Dhillon, Dongarra, Hammarling, Henry, Petitet, Stanley, Walker, and Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997. http://www.netlib.org/scalapack/.
4. F. Desprez, J.J. Dongarra, and B. Tourancheau. Performance Study of LU Factorization with Low Communication Overhead on Multiprocessors. *Parallel Processing Letters*, 5(2):157–169, 1995.
5. F. Desprez and B. Tourancheau. LOCCS: Low Overhead Communication and Computation Subroutines. *Future Generation Computer Systems*, 10:279–284, 1994.
6. Al Geist, Beguelin, Dongarra, Jiang, Manchek, and Sunderam. *PVM, a users' guide and tutorial*. MIT Press, 1994. http://www.netlib.org/pvm .
7. Marc Herbert, Frederic Naquin, Loïc Prylli, Bernard Tourancheau, and Roland Westrelin. Protocole pour le gbit/s en reseau local: l'experience myrinet. *Calculateurs Paralleles, Reseaux et Systemes Repartis*, 1998.

8. L. Giraudeau Georges Perrot S. Petit and B. Tourancheau. 3-d air flow simulation software for paragliders. Technical Report 96-35, LIP-ENS Lyon, 69364 Lyon, France, 1996.
9. J. Katz & A. Plotkin. *Low-speed Aerodynamics From Wing Theory to Panel Methods*. MCGraw-Hill, Inc., 1991.
10. L. Prylli and B. Tourancheau. Efficient block cyclic array redistribution. *Journal of Parallel and Distributed Computing*, (45):63–72, 1997.
11. Loïc Prylli and Bernard Tourancheau. Protocol design for high performance networkin g: Myrinet experience. Technical Report 97-22, LIP-ENS Lyon, 69364 Lyon, France, 1997.
12. Loïc Prylli and Bernard Tourancheau. Bip: a new protocol designed for high performance networking on myrinet. In *Workshop PC-NOW, IPPS/SPDP98*, Orlando, USA, 1998.
13. Y. Robert, B. Tourancheau, and G. Villard. Data allocation strategies for the gauss and jordan algorithms on a ring of processors. *Information Processing Letters*, 31:21–29, 1989.
14. M. Snir, S.W. Otto, S. Huss-Lederman, D.W. Walker, and J.J. Dongarra. Mpi: The complete reference, 1996. http://www.netlib.org/mpi/.
15. Roland Westrelin. Réseaux haut débit et calcul parallèle: étude de myrinet. Master's thesis, LHPC, CPE-Lyon, 1997.

2D Pseudo-Spectral Parallel Navier-Stokes Simulations for Compressible Subsonic Flows

Elisabeth Fournier[1] and Serge Gauthier[1]

CEA/Bruyeres-le-Chatel, BP 12,
91680 Bruyeres-le-Chatel Cedex, FRANCE
Elisabeth.Fournier@bruyeres.cea.fr,
Serge.Gauthier@bruyeres.cea.fr

Abstract. A 2D Fourier-Chebyshev pseudo-spectral method for the full Navier-Stokes equations with a dynamical domain decomposition technique has been parallelized on a SPMD machine, a CRAY-T3E. The parallelism is grounded on the distribution of the data among processors and on the simultaneous computations on each subdomain. The SHMEM paradigm is used for communication between processors. Comparisons between the vectorial and the parallel versions of spectral derivatives with Fourier and Chebyshev expansions are presented. Performances versus the number of processors and collocation points are also studied. Validation of the code has been performed by simulating a subsonic Kelvin-Helmholtz instability and comparing with results obtained on vectorial machines. Performances for two different configurations, the Kelvin-Helmholtz and Rayleigh-Taylor flows, are also presented.

1 Numerical Method

In order to study transition to turbulence, very accurate numerical schemes and large resolution are required [2]. With this aim in view, a sophisticated 2D dynamical multidomain pseudo-spectral code has been developed to simulate viscous compressible flows, and especially the Kelvin-Helmholtz and Rayleigh-Taylor instabilities. The numerical method solves the full 2D Navier-Stokes equations. It uses a Fourier-Chebyshev expansion. The features of our method are as follows [1]:

1. Time marching is done with a semi-implicit third order Runge-Kutta scheme in a low-storage formulation. The advective terms are treated explicitly and all diffusion terms are handled implicitly. Since transport coefficients are constant, the implicit stage is performed in the Fourier space by means of a Chebyshev iterative scheme. This procedure allows us to use larger time steps.
2. The vertical direction is decomposed into non-overlapping subdomains. Density is matched with a simple upwind procedure. Velocities and temperature are handled with the influence matrix method which reflects the continuity of the function and its first normal derivative at the interface.

3. In each subdomain, a self-adaptive transformation of a coordinate is used. Since strong gradients may occur in the middle of the subdomain, a transformation of coordinate is used to bring the mesh points in the vicinity of the gradient. These mappings and the location of the interfaces between subdomains need to be self-adaptive because gradients move in time. Indeed, this is an interesting feature of this numerical method to automatically optimize the locations of the interfaces by minimizing the H^2_ω norm.

2 The Parallelization Method

At first, the numerical code was running on a vectorial computer, a CRAY-YMP. Because the use of this code was limited by the memory size of the computer (the resolution was also restricted) and the time required for a whole simulation, the unique solution was to develop a parallel version for 2D and later 3D calculations. In our case, the parallelization began on a CRAY-T3D, and then finished on a CRAY-T3E. This type of machines offers two advantages : it allows us to increase the total available memory and simultaneously to decrease the cost in CPU time. For example, the maximum allocated memory for an user is $168 * 16$ Mwords on a CRAY-T3E with 168 processor elements, whereas only 512 Mwords are available on a CRAY-T90 (with 24 processors, and sometimes 24 users together!). This new version permits us to execute more voluminous calculations, that means to increase precision and also resolution, *i.e.*, the total number of mesh points. The CRAY-T3E allows only SIMD programming (Single Program Multiple Data), that means that a single program is forked on every processor element (PE), which computes with its own data.

2.1 Domain Decomposition

The numerical method was particularly adapted to parallelism, because it uses a domain decomposition method. Since the physical domain is divided into a little number of subdomains (typically 3 to 9) in one direction (the vertical z-direction), the parallelization procedure concerns the distribution of the subdomains and their physical associated quantities (velocities, density, temperature, pressure, energy ...) on groups of processors. Subdomains are shared out among processors. So, each processor is assigned to a fixed subdomain.

We can represent each physical quantity in a form of a 2D matrix and suppose that columns contain all x-data at fixed z-value, and rows all z-data at fixed x-value. This representation leads us to distribute all columns of a global matrix on group of processors. So, each processor contains a little number of whole columns. In the code, Fast Fourier Transform (FFT) is performed in the x-direction (x-derivative). It can be easily parallelized : each processor performs a few FFTs, while all processors are running in parallel. Serial computations

are transformed into parallel computations. Another current operation is the z-derivative through a matrix-matrix product. Because processors do not possess in their local memory the data needed for this product or any other global operation, data transfer is required, but it suffers from too much communication cost. The matrix-matrix product is mostly used in the code kernel, that is the reason why we tried to minimize this cost, by searching the fastest way to transpose a distributed matrix, for instance.

2.2 Strategy for the Parallelization

Choice between PVM and SHMEM The numerical method requires data transfer between processors, *i.e.*, communication, for matrix operations, such as matrix transposition or the research of a global matrix minimum or maximum, and for matching physical quantities at the domain interfaces. To minimize overhead time due to communication, we looked into the use of two paradigms, PVM and SHMEM. Two criteria were taken into account : portability and rapidity.

First, PVM (Parallel Virtual Machine), a message passing library, was used because of its portability. After a series of tests that revealed lack of efficiency, we added SHMEM (Shared Memory Access Library) instructions with the object to decrease execution time. The SHMEM routines are data passing library routines, which can be used as a replacement for message passing routines. Bandwidth is higher with SHMEM and latency time is lower. In other words, that means that a larger number of elements is transfered between processors within a shorter time. To be precise, latency time is only $1.85\mu s$ on a CRAY-T3D and $1.75\mu s$ on a CRAY-T3E for the `shmem-put`, against respectively $34\mu s$ and $12\mu s$ for PVM on the CRAY-T3D and T3E. Bandwidth is about 120 MBytes/s for SHMEM without stream buffers and only 26 MBytes/s for PVM. SHMEM allows to transfer 5 times more elements in a time 6 times shorter. In conclusion, these routines minimize the overhead associated with data passing requests, maximize bandwidth and minimize data latency.

We compared execution times for a special problem that is characteristic in our resolution method : the matrix transposition. In the code kernel, a FFT is executed first on each processor. Then the global matrix has to be transposed, before the z-derivative is computed through a matrix-matrix product. This matrix transposition requires data transfers. Because the vast majority of the time is spent in this type of operations, total execution times for a FFT, a matrix transposition and a matrix-matrix product have been compared with both of paradigms, PVM and SHMEM.

In Fig.1, comparison between times ratio spent in the code kernel using PVM or SHMEM shows clearly that SHMEM is the fastest : by using the `shmem-iput` or `shmem-ixput` (variants of `shmem-iput`), the time on the CRAY-T3D is at

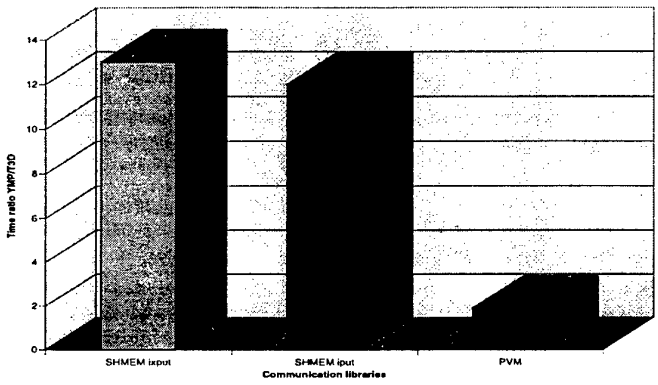


Fig. 1. Ratios of CPU time YMP/T3D vs. the paradigm used, PVM or SHMEM.

least 6 times shorter than with PVM.

Finally, we chose to use only SHMEM because of its efficiency. A next version of MPI-2, a Message Passing Interface, which becomes a standard, will include SHMEM instructions and make easier portability. So, in the near future, both of important criteria will be respected.

Difficulties due to the Implementation of SHMEM SHMEM is a low level library. As a result, it is more efficient than PVM. But it is more difficult to implement SHMEM in a code. Parallelization of a code with SHMEM requires to get in some hardware features. During the parallelization, the programmer has to be very careful because of the following reasons.

On one hand, the data to transfer have usually to be symmetric, that means statically in memory, and more precisely, an object has to be associated to the same address on every PE. This is possible by using a `common` storage area, or using special directives like `!CDIR$ SYMMETRIC`.
A data object is called symmetric, if its local and remote addresses have a known relationship. This is one of the reasons why one has to be very careful with the use of dynamically allocated data objects.

The **shmem-put** and **shmem-get** instructions copy data from memory directly into memory : with the **shmem-put**, data are copied from the local processor memory into the remote processor memory, without advertising the remote processor. The **shmem-get** routine copies data from the remote processor memory into its local memory.
The **shmem-put** call presents some disadvantages: on the CRAY-T3D, cache coherency is only ensured by flushing the cache, because the **shmem-put** routine brings up to date only central memory and not the cache of the remote processor. That is why flushing the cache is very important: otherwise, on the remote processor, cache and memory could contain two different values for the same data! However cache coherency is guaranteed on the CRAY-T3E.

On the other hand, order of data transfered by a **shmem-put** is ensured only by calling the **shmem-fence** routine. Furthermore the **shmem-put** function returns before the end of transfer and poses the problem of asynchronicity. More synchronization between processors has to be explicit and is placed upon the programmer through **shmem-barrier** routines.

All these reasons lead us to use only the **shmem-get** routine, instead of the **shmem-put**, with the intention of avoiding some of these constraints. Whereas the **shmem-put** command is faster than the **shmem-get** one on the CRAY-T3D, performances of both are similar on the CRAY-T3E.

In addition to this, some other collective instructions such as **shmem-broadcast** have been also utilized. With the aim to minimize communication time, another solution is to use PBLAS, a parallel library, which optimizes a lot of operations, like matrix and vector operations. This is under investigation.

3 Tests on the Code Kernel

In a full typical simulation, most of the time is spent in FFT computations, matrix-matrix products and matrix-vector products. To show the efficiency of the parallel version with regard to the vectorial one, we compare the execution times for the code kernel. Such a kernel is defined, on a vectorial computer, as a derivative in the x-direction through a FFT followed by a derivative in the z-direction with a matrix-matrix product. This sequence becomes on a MPP machine a x-derivative (with a FFT), a local matrix transposition and then a z-derivative (with a matrix-matrix product). We applied this kind of calculation to an arbitrary function and measured the total execution time. Each test case has been run 1000 times and the mean value has been calculated.

We carried out a series of tests, by varying the number of mesh points in both directions on one hand, and the number of processors on the other hand. The figures above show the mean time ratio between the CRAY-YMP/T90 and CRAY-T3D/T3E times.

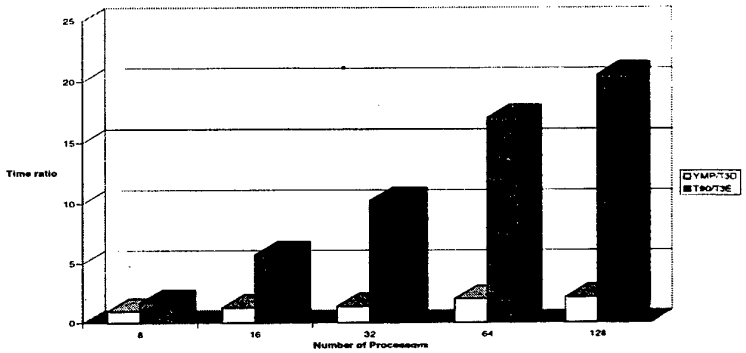


Fig. 2. Ratios of CPU time YMP/T3D and T90/T3E vs. the number of PEs. The number of subdomains is equal to 5. The resolution is 50 z-points in each subdomain and 128 x-points.

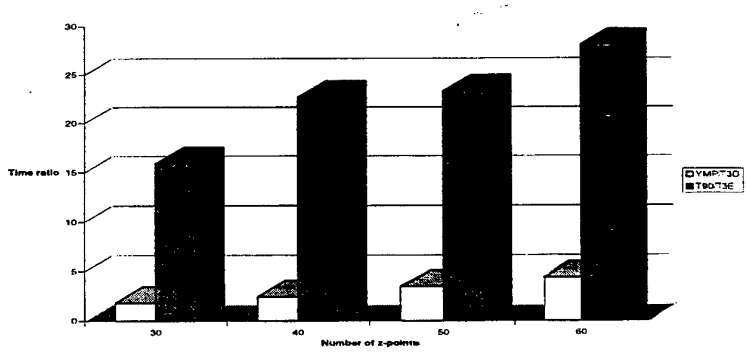


Fig. 3. Ratios of CPU time YMP/T3D and T90/T3E vs. the number of z-points. The number of subdomains is equal to 5. The resolution is 256 z-points.

Figure 2 shows the evolution of this ratio as a function of the number of processors. Figure 3 represents the influence of matrix sizes on the time ratio. This ratio increases with the number of z-points. At a fixed number of z-points.

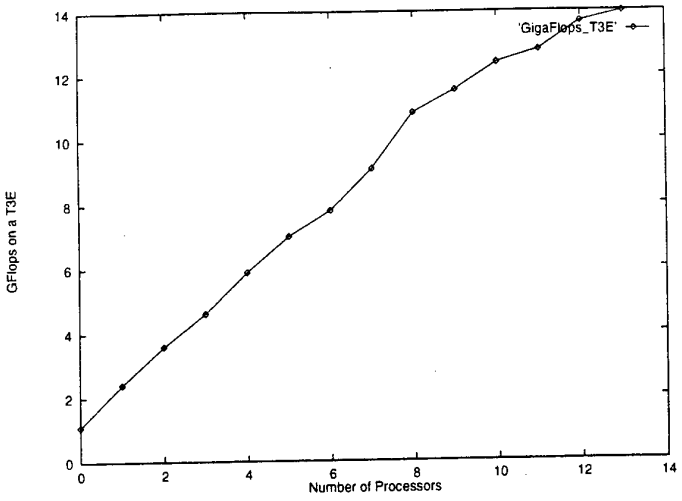


Fig. 4. Performance in terms of GigaFlops on a T3E vs. the number of processors for 9 subdomains and 51 z-points in each.

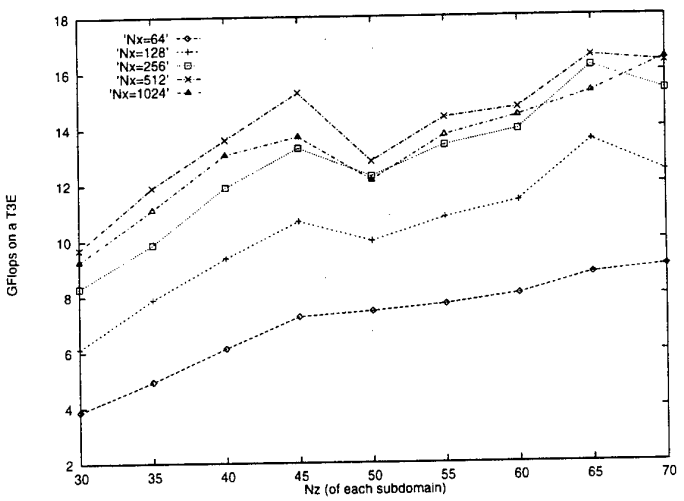


Fig. 5. Performance in terms of GigaFlops on a T3E vs. the number of z-points. The number of x-points is 64, 128, 256, 512 and 1024 respectively.

it grows up with the number of x-points. Moreover, we measured performance, in terms of GigaFlops, in each case. In this case, the physical domain is divided into 9 subdomains with 51 Chebyshev points in the z-direction. Figure 4 represents

the performance obtained on the CRAY-T3E by varying the number of PEs. One can remark that performance increases almost linearly with the number of PEs up to 80 and then saturates, probably because the number of columns in matrix-matrix product is relatively small, for such a number of PEs. In Fig. 5, the curves show the performance, measured in GigaFlops, versus the number of z-points (called N_z). Each curve corresponds to a fixed number of x-points (called N_x). Performance increases with N_z, but not regularly with N_x, and then slightly decreases: the performance maximum is reached for $N_x = 512$. We notice that discrepancies between $N_x = 256$, 512 and 1024 are very small.

4 The Validation of the Parallelization

As first validation of the whole code, we simulated the Kelvin-Helmholtz flow. Its basic state, written in a non-dimensional form is:

$$u = \frac{1}{2}\tanh(2z), v = 0, T = 1 + \frac{\gamma - a}{2}M^2(1 - (2u)^2), P = 1 \qquad (1)$$

with $0 \leq x \leq L_x$ and $-L_z \leq z \leq L_z$. Neumann boundary conditions are applied to the horizontal velocity and the temperature and Dirichlet conditions to the vertical velocity.

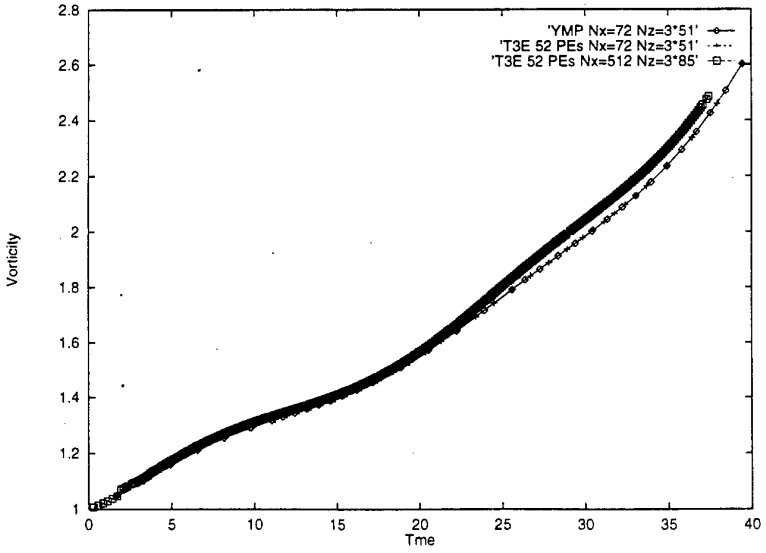


Fig. 6. Evolution in time of the vorticity for the Kelvin-Helmholtz flow. The simulation with the highest resolution is slightly different.

The validation of the parallelization is based on the superposition of both results of the vectorial and parallel versions for a Kelvin-Helmholtz instability [1]. Indeed, we simulated, on both types of machines, the Kelvin-Helmholtz flow with 72 x-points and 155 z-points for 3 subdomains on 52 processors. In Fig. 6, curves represent the evolution in time of the vorticity. A simulation with a higher resolution with 512 x-points and 255 z-points with 5 subdomains on 86 processors has been also performed. This result is very similar to those obtained with 3 subdomains and valid the parallel version.

In Table 1, we compare the Total Execution Time (TET). This time decreases as the number of processors grows up, but this variation is not linear. By comparing the time for the whole simulation on CRAY-YMP and on CRAY-T3E for the Kelvin-Helmholtz instability, we notice that this time is three times shorter on the parallel machine. The efficiency E of a parallel algorithm for a proble instance of size N using P processors is defined by the formula:

$$E(P,N) = \frac{T_1(N)}{P * T_P(N)} \tag{2}$$

where $T_1(N)$ and $T_P(N)$ are respectively the times needed for one processor and P processors.
The efficiency obtained with the whole code is about 0.51 for 10 PEs. At this time, the `f90 -g` compiling option has been used and have slowed down the execution. This can explain a relative bad performance. In this case, communication time is not negligible with respect to the computational time. In addition, the number of mesh points is too small to reach high efficiency.

	T3E 10 PEs	T3E 52 PEs	YMP 1 PE	T3E 1 PE
Total Execution Time in s	$0.398\ 10^4$	$0.183\ 10^4$	$0.474\ 10^4$	$0.179\ 10^5$
Time/node/cycle (in μs)	298	148	406	1530
Efficiency	0.51	0.21		

Table 1. Total Execution Time, time per node per cycle and efficiency for various configurations for the Kelvin-Helmholtz flow.

We also simulated the Rayleigh-Taylor flow. For these simulations, 128 x-points and 5 or 7 subdomains, with 51 z-points in each, were used.
Table 2 contains the times obtained for various configurations for the Rayleigh-Taylor instability. For 5 subdomains (128*255), we obtained an efficiency equal to 0.61, by using the default compiling options. Performances are much better than for the Kelvin-Helmholtz flow. The first reason is the higher number of

	T3E 86 PEs $N_x = 128$ $N_z = 255$	T3E 86 PEs $N_x = 512$ $N_z = 255$	T3E 120 PEs $N_x = 128$ $N_z = 357$	YMP 1 PE $N_x = 128$ $N_z = 255$	T3E 1 PE $N_x = 128$ $N_z = 255$
Time/node/cycle (in μs)	39.9	36.4	63.8	400	2085
Efficiency	0.61				

Table 2. Total Execution Time, time per node per cycle and efficiency for various configurations for the Rayleigh-Taylor flow.

z-points and processors. Because a greater number of calculations are performed within the same time, the TET decreases. As second reason, we can involve the optimization of the parallelization by in-lining, reducing the number of communications, of arrays... The best time obtained for the CRAY-T3E decreases by a factor of 10 in comparison with the CRAY-YMP.

We can expect better performances and efficiency by using other compiling options and further improvements (for instance with the use of PBLAS).

5 Conclusion

We have obtained some results with a parallel version of a 2D pseudo-spectral code using a dynamical domain decomposition method. It solves the full Navier-Stokes equations. The elliptic problems coming from the diffusive terms are solved iteratively in the Fourier space.

The paradigm used for the parallelization is SHMEM, because of its efficiency, in comparison with PVM.
The validation of the parallel version is based, for the Kelvin-Helmholtz instability, on the good superposition of results, that represent the evolution in time of vorticity.
The best efficiency obtained today is equal to 0.61 and the best time per node per cycle is $39.9\mu s$. These performances could be improved.

This work is a successful example of a parallelized pseudo-spectral Fourier-Chebyshev method with a dynamical domain decomposition technique. We are interested in simulation with very high resolution and are applying this method to the Rayleigh-Taylor instability. This sophisticated numerical method coupled with the parallelization will allow us to study in more detail interactions between different modes.[2]

References

1. Renaud F. and Gauthier S.: A Dynamical Pseudo-Spectral Domain Decomposition Technique: Application to Viscous Compressible Flows. J. Comput. Phys.**131** (1997) 89–108
2. Gauthier S., Guillard H., Lumpp T., Male J.M., Peyret R. and Renaud F.: A Spectral Domain Decomposition Technique with Moving Interfaces for Viscous Compressible Flows. Oral communication at ECCOMAS 96. September 1996. Paris.
3. Fournier E. and Gauthier S.: Parallelization of a 2D pseudo-spectral dynamical domain decomposition method for the full Navier-Stokes equations. 6th IWCPTM in Marseille, June 1997.

Key words

Parallelism - Pseudo-spectral methods - Domain decomposition - Kelvin-Helmholtz and Rayleigh-Taylor Instabilities.

A Unified Approach to Parallel Block-Jacobi Methods for the Symmetric Eigenvalue Problem*

D. Giménez[1**], V. Hernández[2***] and A. M. Vidal[2***]

[1] Departamento de Informática y Sistemas. Univ de Murcia.
Aptdo 4021. 30001 Murcia, Spain. (domingo@dif.um.es)
[2] Dpto. de Sistemas Informáticos y Computación. Univ Politécnica de Valencia
Aptdo 22012. 46071 Valencia. Spain. {vhernand,avidal}@dsic.upv.es

Abstract. In this paper we present a unified approach to the design of different parallel block-Jacobi methods for solving the Symmetric Eigenvalue Problem. The problem can be solved designing a logical algorithm by considering the matrices divided into square blocks, and considering each block as a process. Finally, the processes of the logical algorithm are mapped on the processors to obtain an algorithm for a particular system. Algorithms designed in this way for ring, square mesh and triangular mesh topologies are theoretically compared.

1 Introduction

The Symmetric Eigenvalue Problem appears in many applications in science and engineering, and in some cases the problems are of large dimension with high computational cost, therefore it might be better to solve in parallel.

Different approaches can be utilized to solve the Symmetric Eigenvalue Problem on multicomputers:

- The initial matrix can be reduced to condensed form (tridiagonal) and then the reduced problem solved. This is the approach in ScaLAPACK [1].
- A Jacobi method can be used taking advantage of the high level of parallelism of the method to obtain high performance on multicomputers. In addition, the design of block methods allows us to reduce the communications and to use the memory hierarchy better. Different block-Jacobi methods have been designed to solve the Symmetric Eigenvalue Problem or related problems on multicomputers [2, 3, 4, 5].

* The experiments have been performed on the 512 node Paragon on the CSCC parallel computer system operated by Caltech on behalf of the Concurrent Supercomputing Consortium (access to this facility was provided by the PRISM project).
** Partially supported by Comisión Interministerial de Ciencia y Tecnología, project TIC96-1062-C03-02, and Consejería de Cultura y Educación de Murcia, Dirección General de Universidades, project COM-18/96 MAT.
*** Partially supported by Comisión Interministerial de Ciencia y Tecnología, project TIC96-1062-C03-01.

- There are other type of methods in which high performance is obtained because most of the computation is in matrix-matrix multiplications, which can be optimised both in shared or distributed memory multiprocessors. Methods of that type are those based in spectral division [6, 7, 8] or the Yau-Lu method [9].

In this paper a unified approach to the design of parallel block-Jacobi methods is analized.

2 A sequential block-Jacobi method

Jacobi methods work by constructing a matrix sequence $\{A_l\}$ by means of $A_{l+1} = Q_l A_l Q_l^t$, $l = 1, 2, \ldots$, where $A_1 = A$, and Q_l is a plane-rotation that annihilates a pair of nondiagonal elements of matrix A_l. A cyclic method works by making successive sweeps until some convergence criterion is fulfilled. A sweep consists of successively nullifying the $n(n-1)/2$ nondiagonal elements in the lower-triangular part of the matrix. The different ways of choosing pairs (i, j) have given rise to different versions of the method. The odd-even order will be used, because it simplifies a block based implementation of the sequential algorithm, and allows parallelization. With $n = 8$, numbering indices from 1 to 8, and initially grouping the indices in pairs $\{(1,2),(3,4),(5,6),(7,8)\}$, the sets of pairs of indices are obtained as follows:

$$
\begin{array}{ll}
k = 1 & \{(1,2),(3,4),(5,6),(7,8)\} \\
k = 2 & \{2,(1,4),(3,6),(5,8),7\} \\
k = 3 & \{(2,4),(1,6),(3,8),(5,7)\} \\
k = 4 & \{4,(2,6),(1,8),(3,7),5\} \\
k = 5 & \{(4,6),(2,8),(1,7),(3,5)\} \\
k = 6 & \{6,(4,8),(2,7),(1,5),3\} \\
k = 7 & \{(6,8),(4,7),(2,5),(1,3)\} \\
k = 8 & \{8,(6,7),(4,5),(2,3),1\}
\end{array}
$$

When the method converges we have $D = Q_k Q_{k-1} \ldots Q_1 A Q_1^t \ldots Q_{k-1}^t Q_k^t$ and the eigenvalues are the diagonal elements of matrix D and the eigenvectors are the rows of the product $Q_k Q_{k-1} \ldots Q_1$.

The method works over the matrix A and a matrix V where the rotations are accumulated. Matrix V is initially the identity matrix. To obtain an algorithm working by blocks both matrices A and V are divided into columns and rows of square blocks of size $s \times s$. These blocks are grouped to obtain bigger blocks of size $2sk \times 2sk$.

The scheme of an algorithm by blocks is shown in figure 1.

In each block the algorithm works by making a sweep over the elements in the block. Blocks corresponding to the first Jacobi set are considered to have size $2s \times 2s$, adding to each block the two adjacent diagonal blocks. A sweep is performed covering all elements in these blocks and accumulating the rotations to form a matrix Q of size $2s \times 2s$. Finally, the corresponding columns and rows

```
WHILE convergence not reached DO
      FOR every pair (i, j) of indices in a sweep DO
            perform a sweep on the block of size 2s × 2s formed
              by the blocks of size s × s, A_ii, A_ij and A_jj,
              accumulating the rotations on a matrix Q of size 2s × 2s
            update matrices A and V performing matrix-matrix
              multiplications
      ENDFOR
ENDWHILE
```

Fig. 1. Basic block-Jacobi iteration.

of blocks of size $2s \times 2s$ of matrix A and the rows of blocks of matrix V are updated using Q.

After completing a set of blocked rotations, a swap of column and row blocks is performed. This brings the next blocks of size $s \times s$ to be zeroed to the subdiagonal, and the process continues nullifying elements on the subdiagonal blocks.

Because the sweeps over each block are performed using level-1 BLAS, and matrices A and V can be updated using level-3 BLAS, the cost of the algorithm is:

$$8k_3n^3 + (12k_1 - 16k_3)\,n^2s + 8k_3ns^2 \quad flops, \tag{1}$$

when computing eigenvalues and eigenvectors. In this formula k_1 and k_3 represent the execution time to perform a floating point operation using level-1 or level-3 BLAS, respectively.

3 A logical parallel block-Jacobi method

To design a parallel algorithm, what we must do first is to decide the distribution of data to the processors. This distribution and the movement of data in the matrices determine the necessities of memory and data transference on a distributed system. We begin analyzing these necessities considering processes but not processors, obtaining a logical parallel algorithm.

Each one of the blocks of size $2sk \times 2sk$ is considered as a process and will have a particular necessity of memory. At least it needs memory to store the initial blocks, but some additional memory is necessary to store data in sucessive steps of the execution.

A scheme of the method is shown in figure 2. The method using this scheme is briefly explained below.

```
On each process:
WHILE convergence not reached DO
       FOR every Jacobi set in a sweep DO
              perform sweeps on the blocks of size 2s × 2s
                corresponding to indices associated to the processor,
                accumulating the rotations
              broadcast the rotation matrices
              update the part of matrices A and V associated
                to the process, performing matrix-matrix multiplications
              transfer rows and columns of blocks of A and rows
                of blocks of V
       ENDFOR
ENDWHILE
```

Fig. 2. Basic parallel block-Jacobi iteration.

Each sweep is divided into a number of steps corresponding each step to a Jacobi set.

For each Jacobi set the rotations matrices can be computed in parallel, but working only processes associated to blocks $2sk \times 2sk$ of the main diagonal of A. On these processes a sweep is performed on each one of the blocks it contains corresponding to the Jacobi set in use, and the rotations on each block are accumulated on a rotations matrix of size $2s \times 2s$.

After the computation of the rotations matrices, they are sent to the other processes corresponding to blocks in the same row and column in the matrix A, and the same row in the matrix V. And then the processes can update the part of A or V they contain.

In order to obtain the new grouping of data according to the next Jacobi set it is necessary to perform a movement of data in the matrix, and that implies a data transference between processes and additional necessities of memory. It is illustrated in figure 3, where $\frac{n}{s} = 16$, the rows and columns of blocks are numbered from 0 to 15, and the occupation of memory on an odd and an even step is shown. In the figure blocks containing data are marked with $\times$.

Thus, to store a block of matrix A it is necessary to reserve a memory of size $(2sk + s) \times (2sk + s)$, and to store a block of V is necessary a memory of size $(2sk + s) \times 2sk$.

4 Parallel block-Jacobi methods

To obtain parallel algorithms it is necessary to assign each logical process to a processor, and this assignation must be in such a way that the work is balanced.

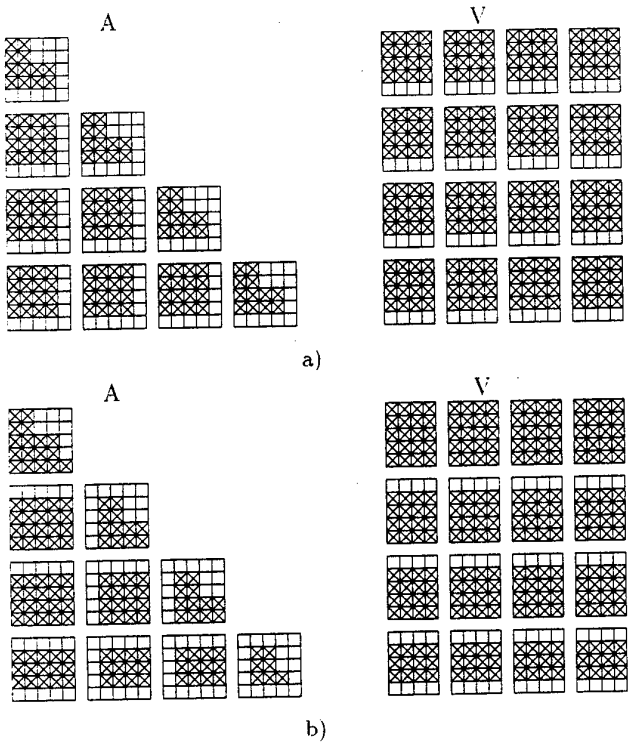


Fig. 3. Storage of data: a) on an odd step, b) on an even step.

The most costly part of the algorithm is the updating of matrices A and V (that produces the cost of order $O(n^3)$). To assign the data in a balanced way it suffices to balance the updating of the matrices only. In the updating of non-diagonal blocks of matrix A, $\frac{n}{2sk} \times \frac{n}{2sk}$ data are updated pre- and post-multiplying by rotations matrices. In the updating of diagonal blocks only elements in the lower triangular part of the matrix are updated pre- and post-multiplying by rotations matrices. And in the updating of matrix V, $\frac{n}{2sk} \times \frac{n}{2sk}$ data are updated but only pre-multiplying by rotations matrices. So, we can see the volume of computation on the updating of matrices on processes corresponding to a non-diagonal block of matrix A is twice that of processes corresponding to a block of V or a diagonal block of A. This must be had in mind when designing parallel algorithms.

An algorithm for a ring Considering $q = \frac{n}{2sk}$ and a ring with $p = \frac{q}{2}$ processors, $P_0, P_1, \ldots, P_{p-1}$, a balanced algorithm can be obtained assigning to each processor P_i, rows i and $q-1-i$ of matrices A and V. So, each processor P_i contains blocks A_{ij}, with $0 \leq j \leq i$, $A_{q-1-i,j}$, with $0 \leq j \leq q-1-i$, and V_{ij}

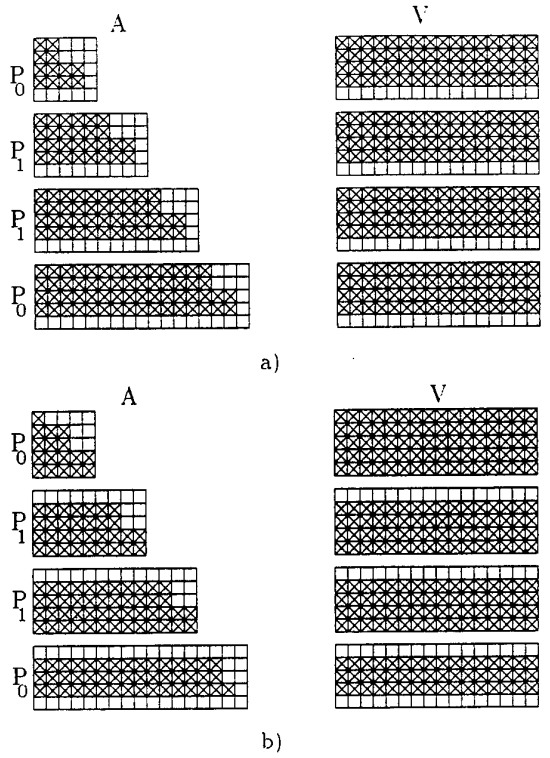


Fig. 4. Storage of data: a) on an odd step, b) on an even step. Algorithm for ring topology.

and $V_{q-1-i,j}$, with $0 \leq j < q$.

To save memory and to improve computation some of the processes in the logical method can be grouped, obtaining on each processor four logical processes, corresponding to the four rows of blocks of matrices A and V contained in the processor. In figure 4 the distribution of data and the memory reserved are shown, for $p = 2$ and $\frac{n}{s} = 16$. In that way, $(2sk+s)(2n+2sk+2s)$ possitions of memory are reserved on each processor.

The arithmetic cost per sweep when computing eigenvalues and eigenvectors is:

$$8k_3\frac{n^3}{p} + (12k_1 - 8k_3)\frac{n^2 s}{p} + 12k_1\frac{ns^2}{p} \quad flops. \tag{2}$$

And the cost per sweep of the communications is:

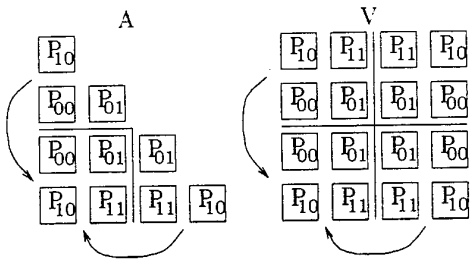


Fig. 5. Folding the matrices in a square mesh topology.

$$\beta(p+3)\frac{n}{s} + \tau\left(8n^2 + 2ns - \frac{2n^2}{p}\right) \quad , \tag{3}$$

where β represents the start-up time, and τ the time to send a double precision number.

An algorithm for a square mesh In a square mesh, a way to obtain a balanced distribution of the work is to fold the matrices A and V in the system of processors, such as is shown in figure 5, where a square mesh with four processors is considered.

To processor P_{ij} are assigned the next blocks: from matrix A block $A_{\sqrt{p}+i,j}$, if $i \leq \sqrt{p}-1-j$ block $A_{\sqrt{p}-1-i,j}$, and if $i \geq \sqrt{p}-1-j$ block $A_{\sqrt{p}+i,2\sqrt{p}-1-j}$; and from matrix V blocks $V_{\sqrt{p}+i,j}$, $V_{\sqrt{p}-1-i,j}$, $V_{\sqrt{p}+i,2\sqrt{p}-1-j}$ and $V_{\sqrt{p}-1-i,2\sqrt{p}-1-j}$. The memory reserved on each processor in the main antidiagonal is $(2sk+s)(14sk+3s)$, and in each one of the other processors $(2sk+s)(12sk+2s)$.

This data distribution produces an inbalance in the computation of the rotations matrices, because only processors in the main antidiagonal of processors work in the sweeps over blocks in the diagonal of matrix A. On the other hand, this inbalance allows us to overlap computations and communications.

The arithmetic cost per sweep when computing eigenvalues and eigenvectors is:

$$8k_3\frac{n^3}{p} + (12k_1 + 2k_3)\frac{n^2 s}{\sqrt{p}} + 12k_1\frac{ns^2}{\sqrt{p}} \quad flops. \tag{4}$$

And the cost per sweep of the communications is:

$$\beta(\sqrt{p}+4)\frac{n}{s} + \tau n^2\left(2 + \frac{7}{\sqrt{p}}\right) \quad . \tag{5}$$

Comparing equations 4 and 2 we can see the arithmetic cost is lower in the algorithm for a ring, but only in the terms of lower order. Furthermore, communications and computations can be overlapped in some parts of the algorithm for a mesh.

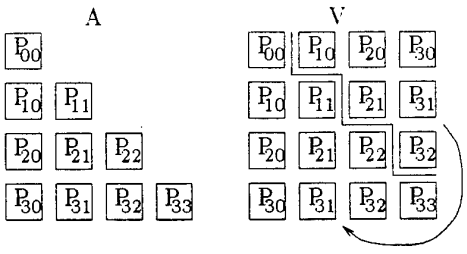


Fig. 6. The storage of matrices in a triangular mesh topology.

An algorithm for a triangular mesh In a triangular mesh, matrix A can be assigned to the processors in an obvious way, and matrix V can be assigned folding the upper triangular part of the matrix over the lower triangular part (figure 6).

To processor P_{ij} $(i \geq j)$ are assigned the blocks A_{ij}, V_{ij} and V_{ji}. The memory reserved on each processor in the main diagonal is $(2sk + s)(4sk + s)$, and in each one of the other processors $(2sk + s)(6sk + s)$.

Only processors in the main diagonal work in the sweeps over blocks in the diagonal of matrix A. In this case, as happens in a square mesh, the inbalance allows us to overlap computations and communications.

If we call r to the number of rows and columns in the processors system, r and p are related by the formula $r = \frac{-1+\sqrt{1+8p}}{2}$, and the arithmetic cost per sweep of the algorithm when computing eigenvalues and eigenvectors is:

$$16k_3\frac{n^3}{r^2} + (12k_1 + 2k_3)\frac{n^2 s}{r} + 12k_1\frac{ns^2}{r} \quad flops. \tag{6}$$

And the cost per sweep of the communications is:

$$\beta(r + 7)\frac{n}{s} + \tau\left(2n^2 + \frac{6n^2}{r} + 4ns\right) \quad . \tag{7}$$

The value of r is a little less than $\sqrt{2p}$. Thus, the arithmetic cost of this algorithm is worse than that of the algorithm for a square mesh, and the same happens with the cost of communications. But when p increases the arithmetic costs tend to be equal, and the algorithm for triangular mesh is better than the algorithm for a square mesh, due to a more regular distribution of data.

5 Comparison

Comparing the theoretical costs of the algorithms studied it is possible to conclude the algorithm for a ring is the best and the algorithm for a triangular mesh

Table 1. Theoretical costs per sweep of the different parts of the algorithms.

	comp. rotations	*update matrices*	*broadcast*	*trans. data*
ring	$8\frac{n^3}{p} - 8\frac{n^2 s}{p}$	$12\frac{n^2 s}{p} + 12\frac{n s^2}{p}$	$(p-1)\frac{n}{s}\beta$ $\left(2n^2 - 2\frac{n^2}{p}\right)\tau$	$4\frac{n}{s}\beta$ $(6n^2 + 2ns)\,\tau$
sq. mesh	$8\frac{n^3}{p} + 2\frac{n^2 s}{\sqrt{p}}$	$12\frac{n^2 s}{\sqrt{p}} + 12\frac{n s^2}{\sqrt{p}}$	$\sqrt{p}\frac{n}{s}\beta$ $2n^2\tau$	$4\frac{n}{s}\beta$ $7\frac{n^2}{\sqrt{p}}\tau$
tr. mesh	$16\frac{n^3}{r^2} + 2\frac{n^2 s}{r}$	$12\frac{n^2 s}{r} + 12\frac{n s^2}{r}$	$r\frac{n}{s}\beta$ $2n^2\tau$	$7\frac{n}{s}\beta$ $\left(6\frac{n^2}{r} + 4ns\right)\tau$

is the worst. This can be true when using a small number of processors, but it is just the opposite when the number of processors and the matrix size increase, due to the overlapping of computations and communications on the algorithms for a mesh.

Some attention has been paid to the optimization of parallel Jacobi methods by overlapping communication and computation [10, 11], and in the mesh algorithms here analysed the inbalance in the computation of rotation matrices makes possible this overlapping. Adding the arithmetic and the communication costs in equations 2 and 3, 4 and 5, and 6 and 7, the total cost per sweep of the algorithms for ring, square mesh and triangular mesh, respectively, can be estimated; but these times have been obtained without regard to the overlapping of computation and communication. In the algorithm for a ring there is practically no overlapping because the computation of rotations is balanced, and after the computation of the rotation matrices each processor is involved in the broadcast, and the updating of matrices begins only after the broadcast finishes. But in the algorithms for mesh the computation of rotations is performed only by the processors in the main diagonal or antidiagonal in the system of processors, and this makes the overlapping possible.

To compare in more detail the three methods, in table 1 the costs per sweep of each part of the algorithms are shown.

The three algorithms have an isoefficiency function $f(n) = p$, but the algorithms for mesh are more scalable in practice. The value of the isoefficiency function appears from the term corresponding to rotations broadcast, which has a cost $O(n^2 p)$, but in the algorithm for a ring this is the real cost, because the matrices A and V can not be updated before the rotations have been broadcast. It is different in the algorithms for a mesh topology, where the execution times obtained are upper-bounds. In these algorithms the rotations broadcast can be overlapped with the updating of the matrices (as shown in [12] for systolic arrays) and when the size of the matrices increases the total cost can be better approximated by adding the costs of table 1 but without the cost of broadcast, which is overlapped with the updating of the matrices. In this way, the isoefficiency function of the algorithms for mesh topology is $f(n) = \sqrt{p}$, and these

methods are more scalable.

In addition, few processors can be utilized efficiently in the algorithm for a ring, for example, with $n = 1024$, if $p = 64$ the block size must be lower or equal to 8, but when using 64 processors on the algorithm for a square mesh the block size must be lower or equal to 64.

The overlapping of communication and computation in the algorithm for triangular mesh is illustrated in figure 7. In this figure matrices A and V are shown, and a triangular mesh with 21 processors is considered. The first steps of the computation are represented writing into each block of the matrices which part of the execution is carried out: R represents computation of rotations, B broadcast of rotation matrices, U matrix updating, and D transference of data. The numbers represent which Jacobi set is involved, and an arrow indicates a movement of data between blocks in the matrices and the corresponding communication of data between processors. We will briefly explain some of the aspects in the figure:

- Step 1: Rotation matrices are computed by the processors in the main diagonal of processors.
- Step 2: The broadcast of rotation matrices to the other processors in the same row and column of processors begins.
- Step 3: Computation and communication are overlapped. All the steps in the figure have not the same cost (the first step has a cost $24k_1\frac{ns^2}{p}$ and the second $2\beta + 4\frac{ns}{p}\tau$), but if a large size of the matrices is assumed the cost of the computational parts is much bigger than that of the communication parts, therefore communication in this step finishes before computation.
- Step 4: More processors begin to compute and the work is more balanced.
- Step 5: Update of matrices has finished in processors in the main diagonal and the subdiagonal of processors, and the movement of rows and columns of blocks begins in order to obtain the data distribution needed to perform the work corresponding to the second Jacobi set.
- Step 6: The computation of the second set of rotation matrices begins in the diagonal before the updating of the matrices have finished. All the processors are involved in this step, and the work is more balanced than in the previous steps. If the cost of computation is much bigger than the cost of communication, the broadcast could have finished and all the processors could be computing.
- Step 7: Only four diagonals of processors can be involved at the same time in communications. Then, if the number of processors increases the cost of comunication becomes less important.
- Step 8: First and second updating are performed at the same time.
- Step 9: After this step all the data have been moved to the positions corresponding to the second Jacobi set.
- Step 10: When the step finishes the third set of rotations can be computed.

As we can see, there is an inbalance at the beginning of the execution, but that is compensated by the overlapping of communication and computation, which

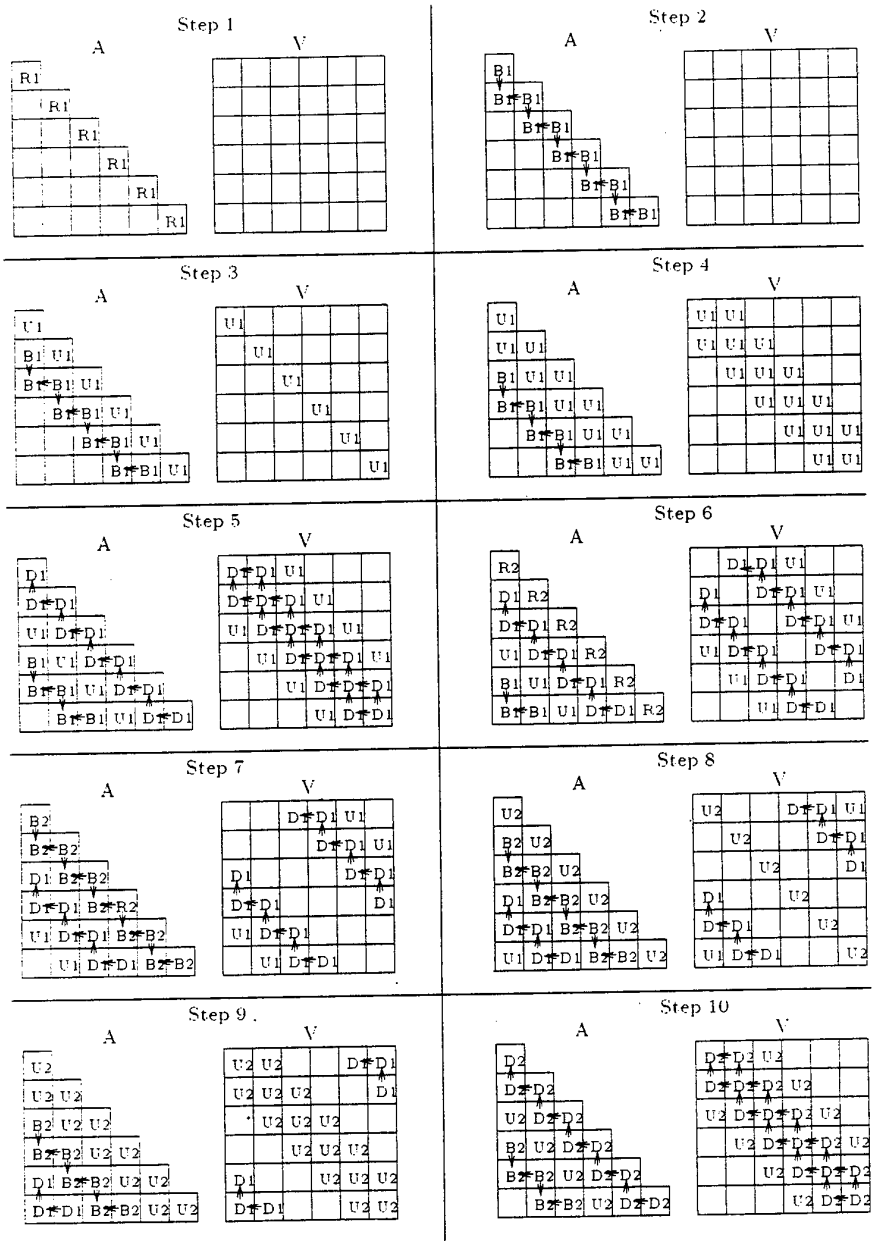


Fig. 7. Overlapping communication and computation on the algorithm for triangular mesh.

Table 2. Comparison of the algorithms for mesh topology. Execution time per sweep (in seconds), on an Intel Paragon.

matrix size :	512		1024	
processors	*triangular*	*square*	*triangular*	*square*
3	10.92		77.61	
4		7.40		47.96
10	3.67		22.71	
16		2.76		15.02
36	1.39		7.27	
64		1.21		5.53
136	0.66		2.74	
256		0.96		2.50

makes it possible to overlook the cost of broadcast to analyze the scalability of the algorithm. The same happens with the algorithm for square mesh.

The algorithm for a triangular mesh is in practice the most scalable due to the overlapping of computation and communication and also to the regular distribution of data, which produces a lower overhead than the algorithm for a square mesh. In tables 2 and 3 the two algorithms for mesh are compared. The results have been obtained on an Intel Paragon XP/S35 with 512 processors. Because the number of processors possible to use is different in both algorithms, the results have been obtained for different numbers of processors. The algorithm for a square mesh has been executed on a physical square mesh, but the algorithm for a triangular mesh has not been executed in a physical triangular mesh, but in a square mesh. This could produce a reduction on the performance of the algorithm for triangular mesh, but this reduction does not happen.

In table 2 the execution time per sweep when computing eigenvalues is shown for matrix sizes 512 and 1024.

In table 3 the Mflops per node obtained with approximately the same number of data per processor are shown. Due to the inbalance in the parallel algorithms the performance is low with a small number of processors, but when the number of processors increases the inbalance is less important and the performance of the parallel algorithms approaches that of the sequential method.

The performance of the algorithm for triangular mesh is much better when the number of processors increases. For example, for a matrix size of 1408, in a square mesh of 484 processors 2.99 Gflops were obtained, while in a triangular mesh of 465 processors 4.23 Gflops were obtained.

6 Conclusions

We have shown how parallel block-Jacobi algorithms can be designed in two steps: first associating one process to each block in the matrices, and then ob-

Table 3. Comparison of the algorithms for mesh topologies. Mflops per node with approximately the same number of data per processor, on an Intel Paragon XP/S35.

data/pro. :	4068		8192		18432		32768	
seq. :	13.10		14.71		18.63		18.39	
proc.	*tr*	*sq*	*tr*	*sq*	*tr*	*sq*	*tr*	*sq*
4		10.52		9.34		15.73		18.14
6	10.95		13.63		14.41		17.00	
15	11.46		12.78		15.53		18.30	
16		10.36		12.16		15.62		17.87
36	10.73	10.21	14.38	12.13	17.04	15.41	18.93	
64		10.15		12.14		15.46		17.65

taining algorithms for a topology by grouping processes and assigning them to processors.

Scalable algorithms have been obtained for mesh topologies and the more scalable in practice is the algorithm for a triangular mesh.

References

1. J. Demmel and K. Stanley. The Performance of Finding Eigenvalues and Eigenvectors of Dense Symmetric Matrices on Distributed Memory Computers. In David H. Bailey, Petter E. Bjørstad, John R. Gilbert, Michael V. Mascagni, Robert S. Schreiber, Horst D. Simon, Virginia J. Torczon and Layne T. Watson, editor, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 528–533. SIAM, 1995.
2. Robert Schreiber. Solving eigenvalue and singular value problems on an undersized systolic array. *SIAM J. Sci. Stat. Comput.*, 7(2):441–451, 1986.
3. Gautam Schroff and Robert Schreiber. On the convergence of the cyclic Jacobi method for parallel block orderings. *SIAM J. Matrix Anal. Appl.*, 10(3):326–346, 1989.
4. Christian H. Bischof. Computing the singular value decomposition on a distributed system of vector processors. *Parallel Computing*, 11:171–186, 1989.
5. D. Giménez, V. Hernández, R. van de Geijn and A. M. Vidal. A block Jacobi method on a mesh of processors. *Concurrency: Practice and Experience*, 9(5):391–411, May 1997.
6. L. Auslander and A. Tsao. On parallelizable eigensolvers. *Ad. App. Math.*, 13:253–261, 1992.
7. S. Huss-Lederman, A. Tsao and G. Zhang. A parallel implementation of the invariant subspace decomposition algorithm for dense symmetric matrices. In *Proceedings Sixth SIAM Conf. on Parallel Processing for Scientific Computing*. SIAM, 1993.
8. Xiaobai Sun. Parallel Algorithms for Dense Eigenvalue Problems. In *Whorkshop on High Performance Computing and Gigabit Local Area Networks, Essen, Germany, 1996*, pages 202–212. Springer-Verlag, 1997.
9. Stéphane Domas and Françoise Tisseur. Parallel Implementation of a Symmetric Eigensolver Based on the Yau and Lu Method. In José M. L. M. Palma

and Jack Dongarra, editor, *Vector and Parallel Processing-VECPAR'96*, pages 140–153. Springer-Verlag, 1997.
10. Makan Pourzandi and Bernard Tourancheau. A Parallel Performance Study of Jacobi-like Eigenvalue Solution. Technical report, 1994.
11. El Mostafa Daoudi and Abdelhak Lakhouaja. Exploiting the symmetry in the parallelization of the Jacobi method. *Parallel Computing*, 23:137–151, 1997.
12. Richard P. Brent and Franklin T. Luk. The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays. *SIAM J. Sci. Stat. Comput.*, 6(1):69–84, 1985.

This article was processed using the LaTeX macro package with LLNCS style

Solving Large-Scale Eigenvalue Problems on Vector Parallel Processors

David L. Harrar II and Michael R. Osborne

Centre for Mathematics and its Applications, School of Mathematical Sciences,
Australian National University, Canberra ACT 0200, Australia
David.Harrar@anu.edu.au,Michael.Osborne@anu.edu.au
WWW home page: http://wwwmaths.anu.edu.au/~dlh and ~mike

Abstract. We consider the development and implementation of eigensolvers on distributed memory parallel arrays of vector processors and show that the concomitant requirements for vectorization and parallelization lead both to novel algorithms and novel implementation techniques. Performance results are given for several large-scale applications and some performance comparisons made with LAPACK and ScaLAPACK.

1 Introduction

Eigenvalue problems (EVPs) arise ubiquitously in the numerical simulations performed on today's high performance computers (HPCs), and often their solution comprises the most computationally expensive algorithmic component. It is imperative that efficient solution techniques and high-quality software be developed for the solution of EVPs on high performance computers.

Generally, specific attributes of an HPC architecture play a significant, if not deterministic, role in terms of choosing/designing appropriate algorithms from which to construct HPC software. There are many ways to differentiate today's HPC architectures, one of the coarsest being *vector* or *parallel*, and the respective algorithmic priorities can differ considerably. However, on some recent HPC architectures – those comprising a (distributed-memory) parallel array of powerful vector processors, e.g., the Fujitsu VPP300 (see Section 5) – it is important, not to focus exclusively upon one or the other, but to strive for high levels of *both* vectorization *and* parallelization.

In this paper we consider the solution of large-scale eigenvalue problems,

$$Au = \lambda u, \tag{1}$$

and discuss how the concomitant requirements for vectorization and parallelization on vector parallel processors has lead both to novel implementations of known methods and to the development of completely new algorithms. These include techniques for both symmetric (or Hermitian) and nonsymmetric A and for matrices with special structure, for example tridiagonal, narrow-banded, etc. Performance results are presented for various large-scale problems solved on a Fujitsu (Vector Parallel Processor) VPP300, and some comparisons are made with analogous routines from the LAPACK [1] and ScaLAPACK [4] libraries.

2 Tridiagonal Eigenvalue Problems

Consider (1) with A replaced by a symmetric, irreducible, tridiagonal matrix T:

$$Tu = \lambda u, \text{ where } T = \text{tridiag}(\beta_i, \alpha_i, \beta_{i+1}), \text{ with } \beta_i \neq 0, \ i = 2, \cdots, n. \quad (2)$$

One of the most popular methods for computing (selected) eigenvalues of T is *bisection* based on the *Sturm sign count*. The widely used LAPACK/Scalapack libraries, for example, each contain routines for (2) which use bisection. Since the techniques are standard, they are not reviewed in detail here (see, e.g., [26]). Instead, we focus on the main computational component: Evaluation of the Sturm sign count, defined by the recursion

$$\sigma_1(\lambda) = \alpha_1 - \lambda, \quad \sigma_i(\lambda) = \left(\alpha_i - \frac{\beta_i^2}{\sigma_{i-1}(\lambda)}\right) - \lambda, \ i = 1, 2, \cdots, n. \quad (3)$$

Zeros of $\sigma_n(\lambda)$ are eigenvalues of T, and the number of times $\sigma_i(\lambda) < 0$, $i = 1, \cdots, n$, is equal to the number of eigenvalues less than the approximation λ.

Vectorization via multisection: In terms of vectorization, the pertinent observation is that the recurrence (3) does not vectorize over the index i. However, if it is evaluated over a sequence of m estimates, λ_j, it is trivial implementationally to interchange i- and j-loops and vectorize over j. This is the basic idea behind *multisection*: An interval containing eigenvalues is split into more than the two subintervals used with bisection. The hope is that the efficiency obtained via vectorization offsets the spurious computation entailed in sign count evaluation for subintervals containing no eigenvalues. For $r > 1$ eigenvalues, another way to vectorize is to bisect r eigenvalue intervals at a time, i.e. *multi-bisection*.[1]

On scalar processors bisection is optimal.[2] On vector processors, however, this is not the case, as shown in an aptly-titled paper [37]: "Bisection is not optimal on vector processors".[3] The non-optimality of bisection on a single VPP300 PE ("processing element") is illustrated in Figure 1 (left), where we plot the time required to compute one eigenvalue of a tridiagonal matrix ($n = 1000$) as a function of m, the number of multisection points (i.e. vector length). The tolerance is $\epsilon = 3 \times 10^{-16}$. For all plots in this section, times are averages from 25 runs.

Clearly, the assertion in [37] holds: Bisection is not optimal. In fact, multisection using up to 3400 points is superior. The minimum time, obtained using 70 points, is roughly 17% the bisection time, i.e. that obtained using LAPACK.

[1] Nomenclaturally, multisecting r intervals might consistently be termed "multi-multisection"; we make no distinction and refer to this also as "multisection".

[2] This is probably not true for most *super*scalar processors: these should be able to take advantage of the chaining/pipelining inherent in multisection.

[3] This may not be true for vector processors of the near future, nor even perhaps all of today's, specifically those with large $n_{1/2}$, compared with, e.g., those in [37]. Additionally, many of today's vector PEs have a separate scalar unit so it is not justifiable to model bisection performance by assuming vector processing with vector length one – bisection is performed by the scalar unit. See the arguments in [10].

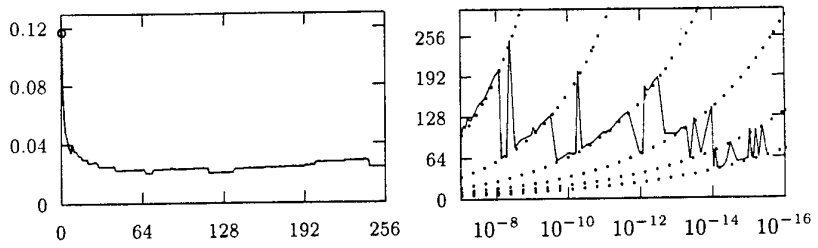


Fig. 1. (Left) Time vs. number of multisection points $m = 1, 256$. (Right) Optimal (i.e., time-minimizing) m vs. accuracy ϵ; dotted lines show $m_{\min}(\epsilon)$ for fixed numbers of multisection steps $\nu = 2, \ldots, 7$ (from left to right).

Once it is decided to use multisection, there still remains a critical question: What is the optimal number of multisection points (equivalently, vector length)? The answer, which depends on r, is discussed in detail in [9]; here we highlight only a few key observations and limit discussion to the case $r = 1$.

Although not noted in [37], the optimal number of points, m_{opt}, depends on the desired accuracy ϵ, as illustrated in Figure 1 (right), where we plot m_{opt} vs. $\epsilon \in [10^{-7}, 3 \times 10^{-16}]$. Note that m_{opt} varies between about 50 and 250.[4] Two effects explain this apparently erratic behavior – one obvious, one not so.

To reach a desired accuracy of ϵ using m multisection points requires

$$\nu = -\lceil \log \epsilon / \log(m+1) \rceil \tag{4}$$

multisection steps. Generally, $m_{\text{opt}}(\epsilon)$ corresponds to some $m_{\min}(\nu, \epsilon)$ at which the ceiling function in (4) causes a change in ν; that is, a minimal number of points effecting convergence to an accuracy of ϵ in ν steps. The dotted lines in Figure 1 (right) indicate $m_{\text{opt}}(\nu, \epsilon)$ for fixed ν. As ϵ is decreased (i.e., moving to the right) $m_{\min}(\nu, \epsilon)$ increases until for some $\epsilon_{\text{crit}}(\nu)$ it entails too much spurious computation in comparison with the (smaller) vector length, $m_{\min}(\nu+1, \epsilon)$, which is now large "enough" to enable adequate vectorization. The optimal now follows along the curve $m_{\min}(\nu + 1, \epsilon)$ until reaching $\epsilon_{\text{crit}}(\nu + 1)$, etc. This explains the occasional transitions from one ν-curve to a $\nu + 1$-curve, but not the oscillatory switching. This is an artifact of a specific performance anomaly associated with the VPP300 processor, as we now elucidate.

In Figure 2 we extend the range of the plot on the left in Figure 1 to include vector lengths up to 2048. The plot "wraps around" in that the bottom line ($i = 0$) is for $m = 1, \ldots, 256$, the second ($i = 1$) for $m = 257, \ldots, 512$, etc. Accuracy is now $\epsilon = 3 \times 10^{-10}$. Note that there is a 10-20% increase in time when the vector length increases from $64i + 8$ to $64i + 9$ (dotted vertical lines) throughout the entire range of vector lengths ($i = 0, \cdots, 31$), though the effect lessens as i increases. This anomalous behavior fosters the erraticity in Figure 1 (right). Whenever $m_{\min}(\nu, \epsilon) = 73, 137, 201, \ldots$ (or one or two more), the time is decreased using $\hat{m}_{\min} = m_{\min}(\hat{\nu}, \epsilon)$ points, where $\hat{\nu} \neq \nu$ is such that $\hat{m}_{\min}$ is not an anomalous vector length.

[4] These values of m_{opt} are roughly five to twenty times those determined in [37], manifesting the effect of the significantly larger $n_{1/2}$ of the VPP300.

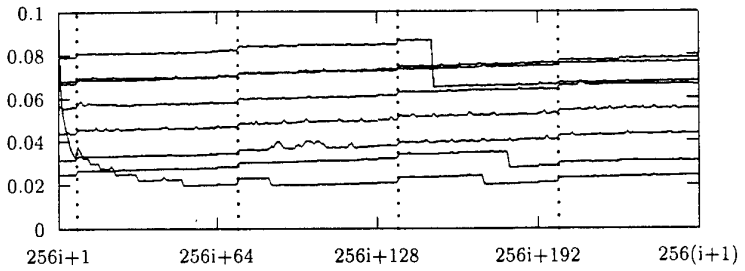


Fig. 2. Time vs. vector length $m = 1, \ldots, 2048$. Dotted lines at $m = 64i + 9$.

This performance anomaly is apparent also for standard vector operations (addition, multiplication, etc.) on the VPP300 [9]; we do not believe it has been noted before, and we are also currently unable to explain the phenomenon. The savings is of course only 10-20% but it is still probably worthwhile to avoid these anomalous vector lengths, in general.

When computing $r > 1$ eigenvalues, the optimal number of points *per eigenvalue interval*, m_{opt}, tends to decrease as r increases, until for some $r = r_b$ multi-bisection is preferable to multisection; this occurs when r is large enough that multi-bisection entails a satisfactory vector length (in relation to $n_{1/2}$). The value of r_b at which this occurs depends on the desired accuracy ϵ. For more details see [9].

Parallelization – Invariant Subspaces for Clustered Eigenvalues: Eigenvectors are calculated using the standard technique of inverse iteration (see, e.g., [14, 40]). Letting λ_i denote a converged eigenvalue, choose u^0 and iterate

$$(T - \lambda_i I)u^k = u^{k-1}, \quad k = 1, 2, \cdots.$$

Generally, one step suffices. Solution of these tridiagonal linear systems is efficiently vectorized via "wrap-around partitioning", discussed in Section 4.

The computation of distinct eigenpairs is communication free. However, computed invariant subspaces corresponding to multiple or tightly clustered eigenvalues are likely to require reorthogonalization. If these eigenvalues reside on different PEs, orthogonalization entails significant communication. Hence, clustered eigenvalues should reside on individual PEs. This can be accomplished in a straightforward manner if complete spectral data is available to all PEs – for example, if the eigenvalue computation is performed redundantly on each PE, or if all-to-all communication is initiated after a distributed computation – in which case redistribution decisions can be made concurrently. However, we insist on a distributed eigenvalue calculation and opine that all-to-all communication is too expensive.

We detect clustering during the refinement process and effect the redistribution in an implicit manner. Once subinterval widths reach a user-defined "cluster tolerance", the smallest and largest sign counts on each PE are communicated to the PE which was initially allocated eigenvalues with those indices, and it is decided which PE keeps the cluster. This PE continues refinement of the clustered eigenvalues to the desired accuracy and computes a corresponding invariant

subspace; the other ignores the clustered eigenvalues and continues refinement of any remaining eigenvalues. If a cluster extends across more than two PEs intermediate PEs are dropped from the computation. Load-imbalance is likely (and generally unavoidable) with the effect worsening as cluster sizes increase. This approach differs from that used in the equivalent ScaLAPACK routines [5].

We note that significant progress has recently been made in the computation of orthogonal eigenvectors for tightly clustered eigenvalues [7, 27], the goal being to obtain these vectors without the need for reorthogonalization. These ideas have not yet been used here, though they may be incorporated into later versions of our routines; it is expected they will be incorporated into future releases of LAPACK and ScaLAPACK [6].

3 Symmetric Eigenvalue Problems

The methods we use for symmetric EVPs entail the solution of tridiagonal EVPs, and this is accomplished using the procedures just described; overall, the techniques are relatively standard and are not discussed in detail.

The first is based on the usual Householder reduction to tridiagonal form, parallelized using a panel-wrapped storage scheme [8]; there is also a version for Hermitian matrices. For sparse matrices we use a Lanczos method [16]. Performance depends primarily on efficient matrix-vector multiplication; the routine uses a diagonal storage format, loop unrolling, and column-block matrix distribution for parallelization. The tridiagonal EVPs that arise are solved redundantly using a single-PE version of the tridiagonal eigensolver. No performance data are presented for the Lanczos solver, but comparisons of the Householder-based routine with those in LAPACK/ScaLAPACK are included in Section 5.

4 Nonsymmetric Eigenvalue Problems

Arnoldi Methods: Arnoldi's method [2] was originally developed to reduce a matrix to upper Hessenberg form; its practicability as a Krylov subspace projection method for EVPs was established in [31]. Letting $V_m = [v_1|\cdots|v_m]$ denote the matrix whose columns are the basis vectors (orthonormalized via, e.g., modified Gram-Schmidt) for the m-dimensional Krylov subspace, we obtain the projected EVP

$$Hy = V_m^* A V_m y = \lambda y, \tag{5}$$

where H is upper Hessenberg and of size $m \ll n$. This much smaller eigenproblem is solved using, e.g., a QR method. Dominant eigenvalues of H approximate those of A with the accuracy increasing with m.

A plethora of modifications can be made to the basic Arnoldi method to increase efficiency, robustness, etc. These include: restarting [31], including the relatively recent implicit techniques [18, 38]; deflation, implicit or explicit, when computing $r > 1$ eigenvalues; preconditioning/acceleration techniques based on.

e.g., Chebyshev polynomials [32], least-squares [33, 34], etc.; spectral transformations for computing non-extremal eigenvalues, e.g., shift-invert [28], Cayley [17, 22], etc.; and of course block versions [35, 36]. For a broad overview see [34].

Our current code is still at a rudimentary stage of development, but we have incorporated a basic restart procedure, shift-invert, and an implicit deflation scheme similar to that outlined in [34] and closely related to that used in [30]. Although it is possible to avoid most complex arithmetic even in the case of a complex shift [28], our routine is currently restricted to real shifts.

In order to be better able to compute multiple or clustered eigenvalues we are also developing a block version. Here matrix-vector multiplication is replaced by matrix-matrix multiplication, leading to another potential advantage, particularly in the context of high performance computing: They enable the use of level-3 BLAS. On some machines this can result in block methods being preferable even in the case of computing only a single eigenvalue [36].

Parallelization opportunities seem to be limited to the reduction phase of the algorithm. Parallelizing, e.g., QR is possible and has been considered by various authors (see, e.g., [3, 13] and the references therein); however, since the projected systems are generally small, it is probably not worthwhile parallelizing their eigensolution. This is the approach taken with P_ARPACK [21], an implementation of ARPACK [19] for distributed memory parallel architectures; although these packages are based on the implicitly restarted Arnoldi method [18, 38], parallelization issues are, for the most part, identical to those for the standard methods. It is probably more worthwhile to limit the maximum Hessenberg dimension to one that is viably solved redundantly on each processor and focus instead on increasing the efficiency of the restarting and deflation procedures and to add some form of preconditioning/acceleration; however, the choices for these strategies should, on vector parallel processors, be predicated on their amenability to vectorization. As mentioned, our code is relatively nascent, and it has not yet been parallelized, nor efficiently vectorized.

Newton-Based Methods: Let $K : C^n \to C^n$ and consider the eigenvalue problem

$$K(\lambda)u = 0, \quad K(\lambda) \equiv (A - \lambda I). \tag{6}$$

(For generalized EVPs, $Au = \lambda Bu$, define $K(\lambda) \equiv (A - \lambda B)$.) Reasonable smoothness of $K(\lambda)$ is assumed but it need not be linear in λ. The basic idea behind using Newton's method to solve EVPs is to replace (6) by the problem of finding zeros of a nonlinear function. Embed (6) in the more general family

$$K(\lambda)u = \beta(\lambda)x, \quad s^*u = \kappa. \tag{7}$$

As λ approaches an eigenvalue $K(\lambda)$ becomes singular so the solution u of the first equation in (7) becomes unbounded for almost all $\beta(\lambda)x$. Hence, the second equation – a scaling condition – can only be satisfied if $\beta(\lambda) \to 0$ as λ approaches an eigenvalue. The vectors s and x can be chosen dynamically as the iteration proceeds; this freedom results in the possibility of exceeding the second-order convergence rate characteristic of Newton-based procedures [25].

Differentiating equations (7) with respect to λ gives

$$K\frac{du}{d\lambda} + \frac{dK}{d\lambda}u = \frac{d\beta}{d\lambda}x, \qquad s^*\frac{du}{d\lambda} = 0.$$

Solving for the Newton correction, the Newton iteration takes the form

$$\lambda \longleftarrow \lambda - \Delta\lambda, \qquad \Delta\lambda = \frac{\beta(\lambda)}{\frac{d\beta}{d\lambda}} = \frac{s^*u}{s^*K^{-1}\frac{dK}{d\lambda}u}.$$

Note that for the non-generalized problem (1), $dK/d\lambda = -I$. The main computational component is essentially inverse iteration with the matrix K; this is effected using a linear solver highly tuned for the VPP300 [23].

Convergence rates, including conditions under which third-order convergence is possible, are discussed in [24]. A much more recent reference is [25] in which the development is completely in terms of generalized EVPs.

Deflation for k converged eigenvalues can be effected by replacing $\beta(\lambda)$ with

$$\phi_k(\lambda) = \frac{\beta(\lambda)}{\prod_{i=1}^{k}(\lambda - \lambda_i)}.$$

However, it is likely to be more beneficial to use as much of the existing spectral information as possible (i.e., not only the eigenvalues). Weilandt deflation (see, e.g., [34]) requires knowledge of left and right eigenvectors, hence involves matrix transposition which is highly inefficient on distributed memory architectures. Hence, we opt for a form of Schur-Weilandt deflation; see [25] for details.

A separate version of the routines for the Newton-based procedures has been developed specifically for block bidiagonal matrices. This algorithm exhibits an impressive convergence rate of 3.56, and uses a multiplicative form of Wielandt deflation so as to preserve matrix structure. Implementationally, inversion of block bidiagonal matrices is required, and this is efficiently vectorized by the technique of *wrap-around partitioning*, which we now describe.

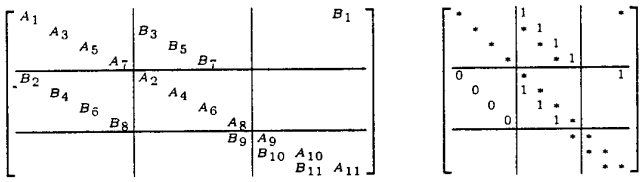
Vectorization – Wrap-Around Partitioning for Banded Systems: Wrap-around partitioning [12] is a technique enabling vectorization of the elimination process in the solution of systems of linear equations. Unknowns are reordered into q blocks of p unknowns each, thereby highlighting groups of unknowns which can be eliminated independently of one another. The natural formulation is for matrices with block bidiagonal (BBD) structure, shown below on the left, but the technique is also applicable to narrow-banded matrices of sufficiently regular structure, as illustrated by reordering a tridiagonal matrix to have BBD form,

$$\begin{bmatrix} A_1 & 0 & 0 & \cdots & B_1 \\ B_2 & A_2 & 0 & \cdots & 0 \\ 0 & B_3 & A_3 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & B_n & A_n \end{bmatrix} \; ; \; \text{tridiag}(\beta_i, \alpha_i, \beta_{i+1}) \longrightarrow \left[\begin{array}{cc|cc|c|cc} 0 & 0 & 0 & 0 & \cdots & \beta_n & \alpha_n \\ \alpha_1 & \beta_2 & 0 & 0 & \cdots & 0 & 0 \\ \hline \beta_2 & \alpha_2 & \beta_3 & 0 & \cdots & 0 & 0 \\ 0 & \beta_3 & \alpha_3 & \beta_4 & \cdots & 0 & 0 \\ \hline \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \hline 0 & 0 & \cdots & \cdots & \beta_{n-2} \; \alpha_{n-2} & \beta_{n-1} & 0 \\ 0 & 0 & \cdots & \cdots & 0 \; \beta_{n-1} & \alpha_{n-1} & \beta_n \end{array}\right].$$

Significant speed-ups – of roughly a factor of 20 over scalar speed – are easily attainable for matrices of size $n > 1000$ in the case that the subblock dimension,

m, is small ($m = 2$ for tridiagonal matrices). The case $q = 2$ corresponds to cyclic reduction, but with wrap-around partitioning p and q need not be exact factors of n; this means that stride-two memory access, which may result in bank conflicts on some computers (e.g., the VPP300), can be avoided. However, q should be relatively small; stride-3 has proven effective on the VPP300. Orthogonal factorization is used rather than Gaussian elimination with partial pivoting since the latter is known to have less favorable stability properties for BBD matrices [41]. Stable factorization preserves BBD structure so that wrap-around partitioning can be applied recursively.

An example is illustrative. Consider a BBD matrix of size $n = 11$. With inexact factors $p = 4$ and $q = 2$ the reordered matrix has the form shown below on the left. In the first stage the blocks B_i, $i = 2, 4, 6, 8$ are eliminated; since each is independent of the others this elimination can be vectorized. Fill-in occurs whether orthogonal factorization or Gaussian elimination with pivoting is used; this is shown at the right, where blocks remaining nonzero are indicated by *, deleted blocks by 0, and blocks becoming nonzero by a 1 indicating fill-in occurred in that position during the first stage.



The potential for recursive application of the reordering is evinced by noting that the trailing block 2×2 submatrix – the one now requiring elimination – is again block bidiagonal. Recursion terminates when the final submatrix is small enough to be solved sequentially.

Arnoldi-Newton Methods: Although the Newton-based procedures ultimately result in convergence rates of up to 3.56, they suffer when good initial data are unavailable; unfortunately this is often the case when dealing with large-scale EVPs. Conversely, Arnoldi methods seem nearly always to move in the right direction at the outset, but may stall or breakdown as the iteration continues, for instance if the maximal Krylov dimension is chosen too small. Heuristics are required to develop efficient, robust Arnoldi eigensolvers. In a sense then, these methods may be viewed as having orthogonal difficulties: Newton methods suffer at the outset, but ultimately perform very well, and Arnoldi methods start off well but perhaps run into difficulties as the iteration proceeds. For this reason we have considered a composition of the two methods: The Arnoldi method is used to get good initial estimates to the eigenvalues of interest and their corresponding Schur vectors for use with the Newton-based procedures. These methods are in the early stages of development.

5 Performance Results on Applications

We now investigate the performance of some of our routines on eigenvalue problems arising in a variety of applications and make some comparisons with corresponding routines from LAPACK and ScaLAPACK. More details and extended performance results on these applications can be found in [11].

Fujitsu VPP300: All performance experiments were performed on the thirteen processor Fujitsu VPP300 located at the Australian National University. The Fujitsu VPP300 is a distributed-memory parallel array of powerful vector PEs, each with a peak rate of 2.2 Gflops and 512 MB or 1.92 GB of memory – the ANU configuration has a peak rate of about 29 Gflops and roughly 14 GB of memory. The PEs consist of a scalar unit and a vector unit with one each of load, store, add, multiply and divide pipes. The network is connected via a full crossbar switch so all processors are equidistant; peak bandwidth is 570MB/s bi-directional. Single-processor routines are written in Fortran 90 and parallel routines in VPP Fortran, a Fortran 90-based language with compiler directives for data layout specification, communication, etc.

Tridiagonal EVP – Molecular Dynamics: First we consider an application arising in molecular dynamics. The multidimensional Schrödinger equation describing the motion of polyatomic fragments cannot be solved analytically. In numerical simulations it is typically necessary to use many hundreds of thousands of basis functions to obtain accurate models of interesting reaction processes; hence, the construction and subsequent diagonalization of a full Hamiltonian matrix, H, is not viable. One frequently adopted approach is to use a Lanczos method, but the Krylov dimension – the size of the resulting tridiagonal matrix, T – often exceeds the size of H. Thus, computation of the eigenvalues of T becomes the dominant computational component.

To test the performance of the tridiagonal eigensolver described in Section 2 we compute some eigenvalues and eigenvectors for a tridiagonal matrix of order $n = 620,000$ arising in a molecular dynamics calculation [29]. In Table 1 we present results obtained using our routines and the corresponding ones from LAPACK on a single VPP300 processor for the computation of one eigenpair and 100 eigenpairs of the 11411 which are of interest for this particular matrix. For further comparison we also include times for the complete eigendecomposition of matrices of size 1000, 3000, and 5000. Values are computed to full machine precision.

# λ	1	100	1000	3000	5000
SSL2VP	23.06	162.7	3.208	23.90	63.43
LAPACK	25.11	768.1	7.783 (25.46)	132.4 (395.5)	581.1 (1717.)

Table 1. Tridiagonal eigensolver: Molecular dynamics application.

For large numbers of eigenvalues the optimal form of multisection is multibisection so that there are no significant performance differences between the two eigenvalue routines. However, the eigenvector routine using wrap-around partitioning is significantly faster than the LAPACK implementation, resulting in considerably reduced times when large numbers of eigenvectors are required. We note that the LAPACK routine uses a tridiagonal QR method when *all* eigenvalues are requested. If $r \leq n-1$ are requested, bisection with inverse iteration is used; the parenthetical times – a factor of three larger – are those obtained computing $n-1$ eigenpairs and serve further to illustrate the efficiency of our implementation. Effective parallelization is evident from the performance results of the symmetric eigensolver, which we next address.

Symmetric EVP – Quantum Chemistry: An application arising in computational quantum chemistry is that of modelling electron interaction in protein molecules. The eigenvalue problem again arises from Schrödinger's equation, $\mathcal{H}\Psi = E\Psi$, where $\mathcal{H}$ is the Hamiltonian operator, E is the total energy of the system, and Ψ is the wavefunction. Using a semi-empirical, as opposed to *ab initio*, formulation we arrive at an eigenvalue problem,

$$F\psi = \epsilon\psi,$$

where ϵ is the energy of an electron orbital and ψ the corresponding wavefunction. The software package MNDO94 [39] is used to generate matrices for three protein structures: (1) single helix of pheromone protein from Euplotes Raikovi (573 atoms, $n = 1482$), (2) third domain of turkey ovomucoid inhibitor (814 atoms, $n = 2068$), and (3) bovine pancreatic ribonuclease A (1856 atoms, $n = 4709$). In Table 2 we give the times required to compute all eigenpairs of the resulting symmetric EVPs using our routines based on Householder reduction and the tridiagonal eigensolver. Also given are times obtained using LAPACK and ScaLAPACK. As noted above, when all eigenvalues are requested these routines use QR on the resulting tridiagonal system and this is faster than their routines based on bisection and inverse iteration. Thus, if fewer than n eigenvalues are requested the performance differences between those routines and ours are amplified considerably.

n	802			2068			4709		
# PEs	1	2	4	1	2	4	1	2	4
SSL2VP(P)	4.130	3.157	1.783	35.55	24.76	12.69	373.1	217.5	113.1
(Sca)LAPACK	5.884	7.362	6.437	68.22	59.62	46.03	694.2	467.8	314.6

Table 2. Symmetric eigensolver: Quantum chemistry application.

The reduction and eigenvector recovery algorithmic components are apparently more efficiently implemented in ScaLAPACK, but the efficiency of our

tridiagonal eigensolvers results in superior performance for the SSL2VP(P) routines. Good scalability of our parallel implementation is also evident.

Nonsymmetric EVP – Optical Physics: Now we consider an application from optical physics, namely the design of dielectric waveguides, e.g., optical fibers. We solve the vector wave equation for general dielectric waveguides which, for the magnetic field components, takes the form of two coupled PDEs

$$\frac{\partial^2 H_x}{\partial x^2} + \frac{\partial^2 H_x}{\partial y^2} - 2\frac{\partial \ln(n)}{\partial y}\left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y}\right) + (n^2k^2 - \beta^2)H_x = 0$$
$$\frac{\partial^2 H_y}{\partial x^2} + \frac{\partial^2 H_y}{\partial y^2} - 2\frac{\partial \ln(n)}{\partial x}\left(\frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y}\right) + (n^2k^2 - \beta^2)H_y = 0.$$

Following [20], a Galerkin procedure is applied to this system, resulting in a coupled system of algebraic equations. Given an optical fiber with indices of refraction n_o and n_i for the cladding and core regions, respectively, eigenvalues of interest correspond to "guided modes" and are given by $\lambda = (\beta/k)^2$, where $\beta/k \in [n_o, n_i)$. The matrix is full, real, and nonsymmetric; despite the lack of symmetry all eigenvalues are real.

Since the Arnoldi-based procedures have not yet been efficiently vectorized/parallelized we do not compare performance against, e.g., ARPACK or P_ARPACK. Instead we illustrate the considerable reduction in time obtained using the Arnoldi method to acquire good initial data for the Newton-based procedures – that is, the Arnoldi-Newton method. This comparison is somewhat contrived since the Newton codes use full complex arithmetic and the matrix for this problem is real. However, it serves to elucidate the effectiveness of combining the two procedures. Considering a fiber with indices of refraction $n_o = 1.265$ and $n_i = 1.415$, we use shift-invert Arnoldi (in real arithmetic) with a shift of $\sigma = 1.79 \in [1.265^2, 1.415^2)$. Once Ritz estimates are $O(1 \times 10^{-9})$, eigenvalues and Schur vectors are passed to the Newton-based routine. Generally, only one Newton step is then necessary to reach machine precision. Times for complex LU factorization are shown, and the number of factorizations required with the Newton and Arnoldi-Newton methods appears in parentheses.

n	cmplx LU	Newton	Arnoldi-Newton
1250	4.105	1414. (108)	328.4 (26)
3200	62.92	20904 (103)	3597. (31)

Table 3. Nonsymmetric eigensolver: Optical physics application.

Matrices are built using the (C++) software library NPL [15] which also includes an eigensolver; our routine consistently finds eigenvalues which NPL's fails to locate – in this example, NPL located ten eigenvalues of interest while our routines find fourteen. The techniques used here are robust and effective. Clearly the use of Arnoldi's method to obtain initial eigendata results in a significant reduction in time. More development work is needed for these methods and for

their implementations, in particular the (block) Arnoldi routine which is not nearly fully optimized.

Complex Nonsymmetric Block Bidiagonal EVP – CFD: Our final application is from hydrodynamic stability; we consider flow between infinite parallel plates. The initial behavior of a small disturbance to the flow is modelled by the Orr-Sommerfeld equation

$$\frac{i}{\alpha R}\left(\frac{d^2}{dz^2}-\alpha^2\right)^2\phi+(U(z)-\lambda)\left(\frac{d^2}{dz^2}-\alpha^2\right)\phi-\frac{d^2U(z)}{dz^2}\phi=0,$$

where α is the wave number, R is the Reynolds' number, and $U(z)$ is the velocity profile of the basic flow; we assume plane Poiseuille flow for which $U(z)=1-z^2$. Boundary conditions are $\phi=d\phi/dz=0$ at solid boundaries and, for boundary layer flows, $\phi\sim 1$ as $z\to\infty$. The differential equation is written as a system of four first-order equations which, when integrated using the trapezoidal rule, yields a generalized EVP

$$K(\alpha,R)u=s^TK(\alpha,R),\quad K(\alpha,R)=A(\alpha,R)-\lambda B,$$

where $A(\alpha,R)$ is complex and block bidiagonal with 4×4 blocks. Further description of the problem formulation can be found in [11].

We compute the neutral curve, i.e. the locus of points in the (α,R)-plane for which $\mathrm{Im}\{c(\alpha,R)\}=0$, using the Newton-based procedures of the last section. The resulting algorithm has convergence rate 3.56 [25]. We use a grid with 5000 points for which A is of order $n=20000$. A portion of the neutral curve is plotted in Figure 3.

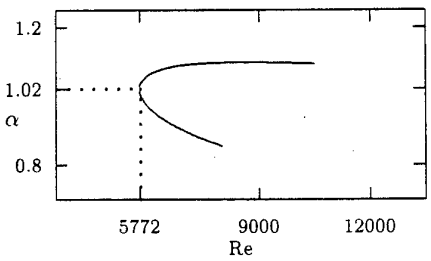


Fig. 3. Complex banded nonsymmetric eigensolver: Neutral stability diagram for Poiseuille flow.

The significant degree of vectorization obtained using wrap-around partitioning enables us to consider highly refined discretizations. Additional results, including consideration of a Blasius velocity profile, can be found in [11].

Acknowledgements

The authors thank the following for assistance with the section on applications: Anthony Rasmussen, Sean Smith, Andrey Bliznyuk, Margaret Kahn, Francois Ladouceur, and David Singleton. This work was supported as part of the Fujitsu-ANU Parallel Mathematical Subroutine Library Project.

References

1. E. ANDERSON, Z. BAI, C. BISCHOF, J. DEMMEL, J. DONGARRA, J. DU CROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNY, S. OSTROUCHOV, AND D. SORENSEN, *LAPACK: Linear Algebra PACKage.* software available from `http://www.netlib.org` under directory "lapack".
2. W. ARNOLDI, *The principle of minimized iterations in the solution of the matrix eigenvalue problem*, Quarterly of Appl. Math., 9 (1951), pp. 17–29.
3. Z. BAI AND J. DEMMEL, *Design of a parallel nonsymmetric eigenroutine toolbox, Part I*, Tech. Rep. Computer Science Division Report UCB/CSD-92-718, University of California at Berkeley, 1992.
4. L. BLACKFORD, J. CHOI, A. CLEARY, E. D'AZEVEDO, J. DEMMEL, I. DHILLON, J. DONGARRA, S. HAMMARLING, G. HENRY, A. PETITET, K. STANLEY, D. WALKER, AND R. WHALEY, *ScaLAPACK: Scalable Linear Algebra PACKage.* software available from `http://www.netlib.org` under directory "scalapack".
5. J. DEMMEL, I. DHILLON, AND H. REN, *On the correctness of some bisection-like parallel eigenvalue algorithms in floating point arithmetic*, Electronic Trans. Num. Anal. (ETNA), 3 (1996), pp. 116–149.
6. I. DHILLON, 1997. Private communication.
7. I. DHILLON, G. FANN, AND B. PARLETT, *Application of a new algorithm for the symmetric eigenproblem to computational quantum chemisty*, in Proc. of the Eight SIAM Conf. on Par. Proc. for Sci. Comput., SIAM, 1997.
8. J. DONGARRA AND R. VAN DE GEIJN, *Reduction to condensed form for the eigenvalue problem on distributed memory architectures*, Parallel Computing, 18 (1992), pp. 973–982.
9. D. HARRAR II, *Determining optimal vector lengths for multisection on vector processors.* In preparation.
10. ——, *Multisection vs. bisection on vector processors.* In preparation.
11. D. HARRAR II, M. KAHN, AND M. OSBORNE, *Parallel solution of some large-scale eigenvalue problems arising in chemistry and physics*, in Proc. of Fourth Int. Workshop on Applied Parallel Computing: PARA'98, Berlin, Springer-Verlag. To appear.
12. M. HEGLAND AND M. OSBORNE, *Wrap-around partitioning for block bidiagonal systems*, IMA J. Num. Anal. to appear.
13. G. HENRY, D. WATKINS, AND J. DONGARRA, *A parallel implemenations of the nonsymmetric QR algorithm for distributed memory architectures*, Tech. Rep. Computer Science Technical Report CS-97-355, University of Tennessee at Knoxville, 1997.
14. I. IPSEN, *Computing an eigenvector with inverse iteration*, SIAM Review. 39 (1997), pp. 254–291.
15. F. LADOUCEUR, 1997. Numerical Photonics Library, version 1.0.
16. C. LANCZOS, *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*, J. Res. Nat. Bur. Standards, 45 (1950), pp. 255–282.
17. R. LEHOUCQ AND K. MEERBERGEN, *Using generalized Cayley transformations within an inexact rational Krylov sequence method*, SIAM J. Mat. Anal. and Appl. To appear.
18. R. LEHOUCQ AND D. SORENSEN, *Deflation techniques for an implicitly restarted Arnoldi iteration*, SIAM J. Mat. Anal. and Appl., 8 (1996), pp. 789–821.

19. R. LEHOUCQ, D. SORENSEN, AND P. VU, *ARPACK: An implementation of the Implicitly Restarted Arnoldi Iteration that computes some of the eigenvalues and eigenvectors of a large sparse matrix*, 1995.
20. D. MARCUSE, *Solution of the vector wave equation for general dielectric waveguides by the Galerkin method*, IEEE J. Quantum Elec., 28(2) (1992), pp. 459–465.
21. K. MASCHOFF AND D. SORENSEN, *P_ARPACK: An efficient portable large scale eigenvalue package for distributed memory parallel architectures parallel supercomputer*, in Proc. of the Third Int. Workshop on Appl. Parallel Comp. (PARA96), Denmark, 1996.
22. K. MEERBERGEN, A. SPENCE, AND D. ROOSE, *Shift-invert and Cayley transforms for detection of rightmost eigenvalues of nonsymmetric matrices*, BIT, 34 (1995), pp. 409–423.
23. M. NAKANISHI, H. INA, AND K. MIURA, *A high performance linear equation solver on the VPP500 parallel supercomputer*, in Proc. Supercomput. '94, 1994.
24. M. OSBORNE, *Inverse iteration, Newton's method, and nonlinear eigenvalue problems*, in The Contributions of J.H. Wilkinson to Numerical Analysis, Symposium Proc. Series No. 19, The Inst. for Math. and its Appl., 1979.
25. M. OSBORNE AND D. HARRAR II, *Inverse iteration and deflation in general eigenvalue problems*, Tech. Rep. Mathematics Research Report No. MRR 012-97, Australian National University. submitted.
26. B. PARLETT, *The Symmetric Eigenvalue Problem*, Prentice Hall, Englewood Cliffs, 1980.
27. B. PARLETT AND I. DHILLON, *Fernando's solution to Wilkinson's problem: an application of double factorization*, Lin. Alg. Appl., 267 (1997), pp. 247–279.
28. B. PARLETT AND Y. SAAD, *Complex shift and invert stategies for real matrices*, Lin. Alg. Appl., 88/89 (1987), pp. 575–595.
29. A. RASMUSSEN AND S. SMITH, 1998. Private communication.
30. A. RUHE, *The rational Krylov algorithm for nonsymmetric eigenvalue problems, III: Complex shifts for real matrices*, BIT, 34 (1994), pp. 165–176.
31. Y. SAAD, *Variations on Arnoldi's method for computing eigenelements of large unsymmetric matrices*, Lin. Alg. Appl., 34 (1980), pp. 269–295.
32. ———, *Chebyshev acceleration techniques for solving nonsymmetric eigenvalue problems*, Math. Comp., 42(166) (1984), pp. 567–588.
33. ———, *Least squares polynomials in the complex plane and their use for solving parse nonsymmetric linear systems*, SIAM J. Numer. Anal., 24 (1987), pp. 155–169.
34. ———, *Numerical Methods for Large Eigenvalue Problems*, Manchester University Press (Series in Algorithms and Architectures for Advanced Scientific Computing), Manchester, 1992.
35. M. SADKANE, *A block Arnoldi-Chebyshev method for computing the leading eigenpairs of large sparse unsymmetric matrices*, Numer. Math., 64 (1993), pp. 181–193.
36. J. SCOTT, *An Arnoldi code for computing selected eigenvalues of sparse real unsymmetric matrices*, ACM Trans. on Math. Soft., 21 (1995), pp. 432–475.
37. H. SIMON, *Bisection is not optimal on vector processors*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 205–209.
38. D. SORENSEN, *Implicit application of polynomial filters in a k-step Arnoldi method*, SIAM J. Mat. Anal. and Appl., 13 (1992), pp. 357–385.
39. W. THIEL, 1994. Program MNDO94, version 4.1.
40. J. WILKINSON, *The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford, 1965.
41. S. WRIGHT, *A collection of problems for which Gaussian elimination with partial pivoting is unstable*, SIAM J. Sci. Stat. Comput., 14 (1993), pp. 231–238.

Solving Eigenvalue Problems on Networks of Processors *

D. Giménez, C. Jiménez, M. J. Majado, N. Marín and A. Martín

Departamento de Informática, Lenguajes y Sistemas Informáticos.
Univ de Murcia. Aptdo 4021. 30001 Murcia, Spain.
{domingo,mmajado,nmarin}@dif.um.es

Abstract. In recent times the work on networks of processors has become very important, due to the low cost and the availability of these systems. This is why it is interesting to study algorithms on networks of processors. In this paper we study on networks of processors different Eigenvalue Solvers. In particular, the Power method, deflation, Givens algorithm, Davidson methods and Jacobi methods are analized using PVM and MPI. The conclusion is that the solution of Eigenvalue Problems can be accelerated by using networks of processors and typical parallel algorithms, but the high cost of communications in these systems gives rise to small modifications in the algorithms to achieve good performance.

1 Introduction

Within the different platforms that can be used to develop parallel algorithms, in recent years special attention is being paid to networks of processors. The main reasons for their use are the lesser cost of the connected equipment, the greater availability and the additional utility as a usual means of work. On the other hand, communications, the heterogeneity of the equipment, their shared use and, generally, the small number of processors used are the negative factors.

The biggest difference between multicomputers and networks of processors is the high cost of communications in the networks, due to the small bandwidth and the shared bus which allows us to send only one message at a time. This characteristic of networks makes it a difficult task to obtain acceptable efficiencies, and also lets one think of the design of algorithms with good performances on a small number of processors, more than of the design of scalable algorithms.

So, despite not being as efficient as supercomputers, networks of processors come up as a new environment to the development of parallel algorithms with a good ratio cost/efficiency. Some of the problems that the networks have can be overlooked using faster networks, better algorithms and new environments appropiate to the network features.

* Partially supported by Comisión Interministerial de Ciencia y Tecnología, project TIC96-1062-C03-02, and Consejería de Cultura y Educación de Murcia, Dirección General de Universidades, project COM-18/96 MAT.

The most used matricial libraries (BLAS, LAPACK [1], ScaLAPACK [2]) are not implemented for those environments, and it seems useful to programme these linear algebra libraries over networks of processors [3, 4, 5, 6, 7]. The implementation of these algorithms can be done over programming environments like PVM [8] or MPI [9], which makes the work easier, although they do not take advantage of all the power of the equipment. The communications libraries utilized have been the free access libraries PVM version 3.4 and MPICH [10] (which is an implementation of MPI) version 1.0.11.

The results obtained using other systems or libraries could be different, but we are interested in the general behaviour of network of processors, and more particularly Local Area Networks (LANs), when solving Linear Algebra Problems. Our intention is to design a library of parallel linear algebra routines for LANs (we could call this library LANLAPACK), and we are working in Linear System Solvers [11] and Eigenvalue Solvers. In this paper some preliminary studies of Eigenvalue Solvers are shown. These problems are of great interest in different fields in science and engineering, and it is possibly better to solve them with parallel programming due to the high cost of computation [12]. The Eigenvalue Problem is still open in parallel computing, where it is necessary to know the eigenvalues efficiently and exactly. For that, we have carried out a study on the development of five methods to calculate eigenvalues over two different environments: PVM and MPI, and using networks with both Ethernet and Fast-Ethernet connections. Experiments have been performed in four different systems:

- A network of 5 SUN Ultra 1 140 with Ethernet connections and 32 Mb of memory on each processor.
- A network of 7 SUN Sparcstation with Ethernet connections and 8 Mb of memory on each processor.
- A network of 12 PC 486, with Ethernet connections, a memory of 8 Mb on each processor, and using Linux.
- A network of 6 Pentiums, with Fast-Ethernet connections, a memory of 32 Mb on each processor, and using Linux.

In this paper we will call these systems SUNUltra, SUNSparc, PC486 and Pentium, respectively.

The approximated cost of floating point operations working with double precision numbers and the theoretical cost of communicating a double precision number, in the four systems, is shown in table 1. The arithmetic cost has been obtained with medium sized matrices (stored in main memory). Also the quotient of the arithmetic cost with respect to the communication cost is shown. These approximations are presented to show the main characteristics of the four systems, but they will not be used to predict execution times because in networks of processors many factors, which are difficult to measure, influence the execution time: collisions when accessing the bus, other users in the system, assignation of processes to processors ... The four systems have also different utilization characteristics: SUNUltra can be isolated to obtain results, but the other three are shared and it is more difficult to obtain representative results.

Table 1. Comparison of arithmetic and communication costs in the four systems utilized.

	floating point cost	*word – sending time*	*quotient*
SUNUltra	0.025 μs	0.8 μs	32
SUNSparc	0.35 μs	0.8 μs	2.28
PC486	0.17 μs	0.8 μs	4.7
Pentium	0.062 μs	0.08 μs	1.29

given v_0
FOR $i = 1, 2, \ldots$
$r_i = Av_{i-1}$
$\beta_i = \| r_i \|_\infty$
$v_i = \frac{r_i}{\beta_i}$
ENDFOR

Fig. 1. Scheme of the sequential Power method.

2 Eigenvalue Solvers

Methods of partial resolution (the calculation of some eigenvalues and/or eigenvectors) of Eigenvalue Problems are studied: the Power method, deflation technique, Givens algorithm and Davidson method; and also the Jacobi method to compute the complete spectrum. We are interested in the parallelization of the methods on networks of processors. Mathematical details can be found in many books ([13, 14, 15, 16]).

Power method. The Power method is a very simple method to compute the eigenvalue of biggest absolute value and the associated eigenvector. Some variations of the method allow us to compute the eigenvalue of lowest absolute value or the eigenvalue nearest to a given number. This method is too slow to be considered as a good method in general, but in some cases it can be useful. In spite of the bad behaviour of the method, it is very simple and will allow us to begin to analyse Eigenvalue Solvers on networks of processors.

A scheme of the algorithm is shown in figure 1. The algorithm works by generating a succession of vectors v_i convergent to an eigenvector q_1 associated to the eigenvalue λ_1, as well as another succession of values β_k convergent to the eigenvalue λ_1. The speed of convergency is proportional to $\frac{\lambda_2}{\lambda_1}$.

Each iteration in the algorithm has three parts. The most expensive is the multiplication matrix-vector, and in order to parallelize the method the attention must be concentrated on that operation. In the parallel implementation, a

```
master:
        given v_0
        FOR i = 1, 2, ...
                broadcast v_{i-1} to the slaves
                receive r_i^(k) from the slaves, and form r_i
                β_i = || r_i ||_∞
                compute norm
                broadcast norm to the slaves
                IF convergence not reached
                        v_i = r_i / β_i
                ENDIF
        ENDFOR

slave k, with k = 0, 1, ..., p - 1:
        FOR i = 1, 2, ...
                receive v_{i-1} from master
                r_i^(k) = A^(k) v_{i-1}
                send r_i^(k) to master
                receive norm from master
        ENDFOR
```

Fig. 2. Scheme of the parallel Power method.

master-slave scheme has been carried out. Matrix A is considered distributed between the slave processes in a block striped partition by rows [17]. A possible scheme of the parallel algorithm is shown in figure 2. The multiplication matrix-vector is performed by the slave processes, but the master obtains β_i and forms v_i. These two operations of cost $O(n)$ could be done in parallel in the slaves, but it would generate more communications, which are very expensive operations in networks of processors.

The arithmetic cost of the parallel method is: $\frac{2n^2}{p} + 3n$ *flops*, and the theoretical efficiency is 100%.

The cost of communications varies with the way in which they are performed. When the distribution of vector v_i and the norm is performed using a broadcast operation, the cost of communications per iteration is: $2\tau_b(p) + \beta_b(p)(n+1) + \tau + \beta n$, where τ and β represent the start-up and the word-sending time, respectively, and $\tau_b(p)$ and $\beta_b(p)$ the start-up and the word-sending time when using a broadcast operation on a system with p slaves. If the broadcasts are replaced by point to point communications the cost of communications per iteration is $\tau(2p+1) + \beta(pn+n+p)$. Which method of communication is preferable depends on the characteristics of the environment (the communication library and the network of processors) we are using.

The parallel algorithm is theoretically optimum if we only consider efficiency, but when studying scalability, the isoefficiency function is $n \propto p^2$, and the scalability is not very good. This bad scalability is caused by the use of a shared bus which avoids sending data at the same time. Also the study of scalability is not useful in this type of system due to the reduced number of processors.

We will experimentally analyse the algorithm in the most and the least adequate systems for parallel processing (Pentium and SUNUltra, respectively). In table 2 the execution time of the sequential and the parallel algorithms on the two systems is shown, varying the number of slaves and the matrix size. Times have been obtained for random matrices, and using PVM and the routine **pvm_mcast** to perform the broadcast in figure 2. The results greatly differ in the two systems due to the big difference in the proportional cost of arithmetic and communication operations. Some conclusions can be obtained:

- Comparing the execution time of the sequential and the parallel algorithms using one slave, we can see the very high penalty of communications in these systems, especially in SUNUltra.
- The best execution times are obtained with a reduced number of processors because of the high cost of communications, this number being bigger in Pentium. The best execution time for each system and matrix size is marked in table 2.
- The availability of more potential memory is an additional benefit of parallel processing, because it allows us to solve bigger problems without swapping. For example, the parallel algorithm in SUNUltra is quicker than the sequential algorithm only when the matrix size is big and the sequential execution produces swapping.
- The use of more processes than processors produces in Pentium with big matrices better results than one process per processor, and this is because communications and computations are better overlapped.

The basic parallel algorithm (figure 2) is very simple but it is not optimized for a network of processors. We can try to improve communications in at least two ways:

- The broadcast routine is not optimized for networks, and it could be better if we replace this routine by point to point communications.
- The diffusion of the norm and the vector can be assembled in only one communication. In that way more data are transferred because the last vector need not be sent, but less communications are performed.

In table 3 the execution time of the basic version (version 1), the version with point to point communications (version 2), and the version where the diffusion of norm and vector are assembled (version 3) are compared. Versions 2 and 3 reduce the execution time in both systems, and the reduction is clearer in SUNUltra because of the bigger cost of communication in this system.

Until now, the results shown have been those obtained with PVM, but the use of MPI produces better results (obviously it depends on the versions we are using). In table 4 the execution time obtained with the basic version of the

Table 2. Execution time (in seconds) of the Power method using PVM, varying the number of processors and the matrix size.

	sequential	1 slave	2 slaves	3 slaves	4 slaves	5 slaves	6 slaves	7 slaves	8 slaves
SUNUltra									
300	**0.069**	0.149	0.255	0.289	0.381	0.547	0.526	0.552	0.638
600	**0.292**	0.497	0.450	0.491	0.629	0.864	0.838	0.849	0.954
900	**0.599**	0.882	0.679	0.735	1.065	1.491	1.457	1.284	1.603
1200	4.211	7.582	1.277	**1.062**	1.426	1.722	1.940	2.004	
1500	23.613	54.464	**1.481**	1.796	1.901	2.233	2.324	2.421	
Pentium									
300	**0.172**	0.281	0.250	0.188	0.292	0.323	0.407	0.405	0.436
600	0.592	1.153	0.639	**0.569**	0.608	0.682	0.599	0.662	0.656
900	1.302	1.762	1.903	1.272	0.908	0.892	**0.842**	0.891	1.171
1200	2.138	8.141	1.544	1.750	1.568	1.224	1.275	1.231	**1.169**
1500	3.368	254.42	2.776	1.904	4.017	2.431	1.911	1.750	**1.680**

Table 3. Comparison of the execution time of the Power method using PVM, with different communication strategy.

	2 slaves			3 slaves			4 slaves		
	ver 1	ver 2	ver 3	ver 1	ver 2	ver 3	ver 1	ver 2	ver 3
SUNUltra									
300	0.255	0.187	**0.168**	0.289	0.262	**0.187**	0.381	0.339	**0.203**
600	0.450	0.418	**0.363**	0.491	0.430	**0.386**	0.629	0.605	**0.341**
900	0.679	0.570	**0.566**	0.735	0.666	**0.567**	1.065	0.688	**0.542**
1200	1.277	1.089	**0.877**	1.062	1.045	**0.854**	1.426	0.949	**0.911**
Pentium									
300	0.250	**0.157**	0.271	0.188	0.188	**0.141**	0.292	0.294	**0.192**
600	0.639	**0.574**	0.585	0.569	**0.516**	0.599	0.608	0.603	**0.557**
900	1.903	**1.879**	2.714	1.272	**0.983**	1.090	0.908	**0.813**	0.985

programme using MPI on SUNUltra is shown. Comparing this table with table 2 we can see the programme with MPI works better when the number of processors increases.

Deflation technique. Deflation technique is used to compute the next eigenvalue and its associated eigenvector starting from a previously known one. This technique is used to obtain some eigenvalues and it is based on the transformation of the initial matrix to another one that has got the same eigenvalues, replacing λ_1 by zero.

To compute **numEV** eigenvalues of the matrix A, the deflation technique can

Table 4. Execution time (in seconds) of the Power method using MPI, varying the number of processors and the matrix size, on SUNUltra.

	1 slave	2 slaves	3 slaves	4 slaves	5 slaves	6 slaves
300	0.15	**0.14**	0.22	0.28	0.27	0.38
600	0.39	**0.31**	**0.31**	0.37	0.55	0.62
900	0.73	0.52	**0.47**	0.56	0.61	0.66
1200	3.58	0.76	**0.70**	0.71	0.79	0.93
1500	22.16	1.37	**0.97**	0.99	1.06	0.94

$A_1 = A$
FOR $i = 1, 2, \ldots, numEV$
  compute by the Power method λ_i **and** $q_i^{(i)}$
  update matrix A: $A_{i+1} = B_{i+1} A_i$
  compute q_i **from** $q_i^{(i)}$
ENDFOR

Fig. 3. Scheme of the deflation technique.

be used performing **numEV** steps (figure 3), computing in each step, using the Power method, the biggest eigenvalue (λ_i) and the corresponding eigenvector ($q_i^{(i)}$) of a matrix A_i, with $A_1 = A$. Each matrix A_{i+1} is obtained from matrix A_i using $q_i^{(i)}$, which is utilized to form matrix B_{i+1}:

$$B_{i+1} = \begin{bmatrix} 1 & 0 & \cdots & 0 & -q_1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & -q_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & -q_{k-1} & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & -q_{k+1} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & -q_n & 0 & \cdots & 1 \end{bmatrix}$$

where $q_i^{(i)'} = (q_1, \ldots, q_{k-1}, 1, q_{k+1}, \ldots, q_n)$.

The eigenvalue λ_i is the biggest eigenvalue in absolute value of matrix A_i, and it is also the i-th eigenvalue in absolute value of matrix A. The eigenvector q_i associated to the eigenvalue λ_i in matrix A is computed from the eigenvector $q_i^{(i)}$ associated to λ_i in the matrix A_i. This computation is performed repeatedly applying the formula $q_i^{(w)} = q_i^{(w+1)} + \frac{a_k^{(w)} q_i^{(w+1)}}{\lambda_i - \lambda_w} q_w^{(w)}$, where $a_k^{(w)}$ is the k-th

Table 5. Execution time (in seconds) of the deflation method using PVM, varying the number of processors and the matrix size, when computing 5% of the eigenvalues.

	sequential	1 slave	2 slaves	3 slaves	4 slaves	5 slaves
SUNUltra						
300	**11.4**	29.6	25.2	28.5	32.4	33.7
600	**184.7**	308.3	239.7	252.1	244.2	273.2
900	914.4	1258.3	862.5	826.6	**765.7**	831.0
Pentium						
300	25.0	32.4	29.9	23.0	22.4	**22.2**
600	407.0	462.4	288.9	239.5	195.8	**194.6**
900	2119.0	2316.5	1460.3	993.7	823.2	**720.7**

column of matrix A_w, with k the index where $q_w^{(w)} = 1$.

The most costly part of the algorithm is the application of the Power method, which has a cost of order $O(n^2)$ per iteration. Therefore, the previously explained Power method can be applied using a scheme master-slave. The cost of the deflation part (update matrix) is $2n^2$ flops, and the cost of the computation of q_i depends on the step and is $5n(i-1)$ flops. These two parts can be performed simultaneously in the parallel algorithm: the master process computes q_i while the slaves processes update matrix A_i. The deflation part of the parallel algorithm (update matrix and compute q_i) is not scalable if a large number of eigenvalues are computed. As we have previously mentioned, scalability is not very important in these types of systems. In addition, this method is used to compute only a reduced number of eigenvalues, due to its high cost when computing a large number of them.

In table 5 the execution time of the sequential and parallel algorithms are shown on SUNUltra and Pentium, using PVM and the basic version of the Power method. Compared with table 2, we can see the behaviour of the parallel Power and deflation algorithms is similar, but that of the deflation technique is better, due to the high amount of computation, the work of the master processor in the deflation part of the execution and the distribution of data between processes in the parallel algorithm.

Davidson method. The Power method is a very simple but not very useful method to compute the biggest eigenvalue of a matrix. Other more efficient methods, as for example Davidson methods [18] or Conjugate Gradient methods [19, 20], can be used.

The Davidson algorithm lets us compute the highest eigenvalue (in absolute value) of a matrix, though it is especially suitable for large, sparse and symmetric matrices. The method is valid for real as well as complex matrices. It works by building a sequence of search subspaces that contain, in the limit, the desired eigenvector. At the same time these subspaces are built, so approximations to the desired eigenvector in the current subspace are also built.

```
V_0 = []
given v_1
k = 1
WHILE convergence not reached
        V_k = [V_{k-1}|v_k]
        orthogonalize V_k using modified Gram-Schmith
        compute H_k = V_k^c A V_k
        compute the highest eigenpair (θ_k, y_k) of H_k
        obtain the Ritz vector u_k = V_k y_k
        compute the residual r_k = A u_k - θ_k u_k
        IF k = k_max
                reinitialize
        ENDIF
        obtain v_{k+1} = (θ_k I - D)^{-1} r_k
        k = k + 1
ENDWHILE
```

Fig. 4. Scheme of the sequential Davidson method.

Figure 4 shows a scheme of a sequential Davidson method. In successive steps a matrix V_k with k orthogonal column vectors is formed. After that, matrix $H_k = V_k^c A V_k$ is formed and the biggest eigenvalue in absolute value (θ_k) and its associated eigenvector (y_k) are computed. This can be done using the Power method, because matrix H_k is of size $k \times k$ and k can be kept small using some reinitialization strategy (when $k = k_{max}$ the process is reinitialized). The new vector v_{k+1} to be added to the matrix V_k to form V_{k+1} can be obtained with the succession of operations: $u_k = V_k y_k$, $r_k = A u_k - \theta_k u_k$ and $v_{k+1} = (\theta_k I - D)^{-1} r_k$, with D the diagonal matrix which has in the diagonal the diagonal elements of matrix A.

To obtain a parallel algorithm the cost of the different parts in the sequential algorithm can be analysed. The only operations with cost $O(n^2)$ are two matrix-vector multiplications: AV_k in the computation of H_k and Au_k in the computation of the residual. AV_k can be accomplished in order $O(n^2)$ because it can be decomposed as $[AV_{k-1}|Av_k]$, and AV_{k-1} was computed in the previous step. The optimum value of k_{max} varies with the matrix size and the type of the matrix, but it is small and it is not worthwhile to parallelize the other parts of the sequential algorithm. Therefore, the parallelization of the Davidson method is done basically in the same way as the Power method: parallelizing matrix-vector multiplications.

This method has been parallelized using a master-slave scheme (figure 5), working all the processes in the two parallelized matrix-vector multiplications, and performing the master process non parallelized operations. In that way,

```
master:
        V_0 = []
        given v_1
        k = 1
        WHILE convergence not reached
                V_k = [V_{k-1} | v_k]
                orthogonalize V_k using modified Gram-Schmith
                send v_k to slaves
                in parallel compute Av_k and accumulate in the master
                compute H_k = V_k^c (AV_k)
                compute the highest eigenpair (θ_k, y_k) of H_k
                obtain the Ritz vector u_k = V_k y_k
                send u_k to slaves
                in parallel compute Au_k and accumulate in the master
                compute the residual r_k = Au_k - θ_k u_k
                IF k = k_max
                        reinitialize
                ENDIF
                obtain v_{k+1} = (θ_k I - D)^{-1} r_k
                k = k + 1
        ENDWHILE

slave k, with k = 1,...,p-1:
        WHILE convergence not reached
                receive v_k from master
                in parallel compute Av_k and accumulate in the master
                receive u_k from master
                in parallel compute Au_k and accumulate in the master
                IF k = k_max
                        reinitialize
                ENDIF
                k = k + 1
        ENDWHILE
```

Fig. 5. Scheme of the parallel Davidson method.

matrix A is distributed between the processes and the result of the local matrix-vector multiplications is accumulated in the master and distributed from the master to the other processes.

Because the operations parallelized are matrix-vector multiplications, as in the Power method, the behaviour of the parallel algorithms must be similar, but better results are obtained with the Davidson method because in this case the master process works in the multiplications.

Table 6 shows the execution time of 50 iterations of the Davidson method

Table 6. Execution time of the Davidson method using MPI, varying the number of processors and the matrix size, on PC486.

	sequ	p=2	p=3	p=4	p=5	p=6	p=7	p=8
300	0.048	0.051	0.048	0.045	0.043	**0.041**	**0.041**	0.042
600		0.129	0.097	0.083	0.076	0.072	0.065	**0.064**
900				0.165	0.131	0.118	0.119	**0.112**

for symmetric complex matrices on PC486 and using MPI, varying the number of processors and the matrix size.

Givens algorithm or bisection. As we have seen in previous paragraphs, on parallel Eigenvalue Solvers whose cost is of order $O(n^2)$ only a small reduction in the execution time can be achieved in networks of processors in some cases: when the matrices are big or the quotient between the cost of communication and computation is small.

In some other Eigenvalue Solvers the behaviour is slightly better. For example, the bisection method is an iterative method to compute eigenvalues in an interval or the k biggest eigenvalues. It is applicable to symmetric tridiagonal matrices. This method is especially suitable to be parallelized, due to the slight communication between processes, which factor increases the total time consumed in a network. When computing the eigenvalues in an interval, the interval is divided in subintervals and each process works in the computation of the eigenvalues in a subinterval. When computing the k biggest eigenvalues, each slave knows the number of eigenvalues it must compute. After that, communications are not necessary but inbalance is produced by the distribution of the spectrum. More details on the parallel bisection method are found in [21].

The eigenvalues are computed by the processes performing successive iterations, and each iteration has a cost of order $O(n)$. Despite the low computational cost good performance is achieved because communications are not necessary after the subintervals are broadcast. Table 7 shows the efficiency obtained using this method to calculate all the eigenvalues or only 20% of them, on SUNUltra and SUNSparc for matrix size 100. The efficiencies are clearly better than in the previous algorithms, even with small matrices and execution time.

Jacobi method. The Jacobi method for solving the Symmetric Eigenvalue Problem works by performing successive sweeps, nullifying once on each sweep the $n(n-1)/2$ nondiagonal elements in the lower-triangular part of the matrix.

It is possible to design a Jacobi method considering the matrix A of size $n \times n$, dividing it into blocks of size $t \times t$ and doing a sweep on these blocks, regarding each block as an element. The blocks of size $t \times t$ are grouped into blocks of size $2kt \times 2kt$ and these blocks are assigned to the processors in such a way that the load is balanced. Parallel block Jacobi methods are explained in more detail in [22].

Table 7. Efficiency of the Givens method using PVM, varying the number of processors, on SUNUltra and SUNSparc, with matrix size 100.

	$p=2$	$p=3$	$p=4$	$p=5$	$p=2$	$p=3$	$p=4$	$p=5$
	all the eigenvalues				*20% of the eigenvalues*			
SUNUltra	0.72	0.63	0.55	0.49	0.52	0.38	0.26	0.19
SUNSparc	1.17	0.94	0.60	0.64	0.81	0.61	0.36	0.37

Table 8. Theoretical speed-up of the Jacobi method, varying the number of processes and processors.

	$p=2$	$p=3$	$p=4$	$p=5$	$p=6$	$p=7$	$p=8$	$p=9$	$p=10$
3	2	2	2	2	2	2	2	2	2
6		3	3	4.5	4.5	4.5	4.5	4.5	4.5
10			4	4	5.3	5.3	8	8	8

In order to obtain a distribution of data to the processors, an algorithm for a logical triangular mesh can be used. The blocks of size $2kt \times 2kt$ must be assigned to the processors in such a way that the work is balanced. Because the most costly part of the execution is the updating of the matrix, and nondiagonal blocks contain twice more elements to be nullified than diagonal blocks, the load of nondiagonal blocks can be considered twice the load of diagonal blocks.

Table 8 shows the theoretical speed-up of the method when logical meshes of 3, 6 or 10 processes are assigned to a network, varying the number of processors in the network from 2 to 10. Higher theoretical speed-up is obtained increasing the number of processes. This produces better balancing, but also more communications and not always a reduction of the execution time. It can be seen in table 9, where the execution time per sweep for matrices of size 384 and 768 is shown, varying the number of processors and processes. The shown results have been obtained in PC486 and SUNUltra using MPI, and the good behaviour of the parallel algorithm is observed because a relatively large number of processors can be used reducing the execution time.

3 Conclusions

Our goal is to develop a library of linear algebra routines for LANs. In this paper some previous results on Eigenvalue Solvers are shown. The characteristics of the environment propitiate small modifications in the algorithms to adapt them to the system. In these environments we do not generally have many processors, and also, when the number of processors goes up, efficiency unavoidably goes down. For these reasons, when designing algorithms for networks of processors it is preferable to think on good algorithms for a small number of processors,

Table 9. Execution time per sweep of the Jacobi method on PC486 and SUNUltra using MPI.

	p=1	p=2	p=3	p=4	p=5	p=6	p=7	p=8	p=9	p=10
SUNUltra: 384 (non swapping)										
1	6.25									
3		4.78	**4.55**							
6			7.08	7.31	8.26					
SUNUltra: 768 (non swapping)										
1	50.78									
3		28.37	**25.50**							
6			39.51	33.11	32.18					
PC486: 384 (non swapping)										
1	43.26									
3		30.32	26.81							
6			28.01	23.15	19.81	18.53				
10				42.20	27.15	30.79	21.46	19.82	**15.49**	17.44
PC486: 768 (swapping)										
1	698.8									
3		217.4	195.6							
6			187.2	142.1	111.6	104.6				
10				158.9	145.6	106.5	96.9	83.5	76.5	**72.5**

and not on scalable algorithms. Because of the great influence of the cost of communications, a good use of the available environments -MPI or PVM- is essential.

References

1. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen. *LAPACK Users' Guide*. SIAM, 1992.
2. ScaLAPACK User's Guide. 1996.
3. L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker and R. C. Whaley. ScaLAPACK: A Linear Algebra Library for Message-Passing Computers. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing. CD-ROM*. SIAM, 1997.
4. J. Demmel and K. Stanley. The Performance of Finding Eigenvalues and Eigenvectors of Dense Symmetric Matrices on Distributed Memory Computers. In D. H. Bailey, P. E. Bjorstad, J. R. Gilbert, M. V. Mascagni, R. S. Schreiber, H. D. Simon, V. J. Torczon and L. T. Watson, editor, *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 528–533. SIAM, 1995.

5. D. Giménez, M. J. Majado and I. Verdú. Solving the Symmetric Eigenvalue Problem on Distributed Memory Systems. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications. PDPTA '97*, pages 744–747, 1997.
6. Kuo-Chan Huang, Feng-Jian Wang and Pei-Chi Wu. Parallelizing a Level 3 BLAS Library for LAN-Connected Workstations. *Journal of Parallel and Distributed Computing*, 38:28–36, 1996.
7. Gen-Ching Lo and Yousef Saad. Iterative solution of general sparse linear systems on clusters of workstations. May 1996.
8. A. Geist, A. Begelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam. Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.. The MIT Press, 1995.
9. Message Passing Interface Forum. A Message-Passing Interface Standard. *International Journal of Supercomputer Applications*, (3), 1994.
10. Users guide to mpich. preprint.
11. F. J. García and D. Giménez. Resolución de sistemas triangulares de ecuaciones lineales en redes de ordenadores. Facultad de Informática. Universidad de Murcia. 1997.
12. A. Edelman. Large dense linear algebra in 1993: The parallel computing influence. *The International Journal of Supercomputer Applications*, 7(2):113–128, 1993.
13. G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1989. Segunda Edición.
14. L. N. Trefethen and D. Bau III. *Numerical Linear Algebra*. SIAM, 1997.
15. David S. Watkins. *Matrix Computations*. John Wiley & Sons, 1991.
16. J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Clarendon Press, 1965.
17. V. Kumar, A. Grama, A. Gupta and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms*. The Benjamin Cummings Publishing Company, 1994.
18. E. R. Davidson. The Iterative Calculation of a Few of the Lowest Eigenvalues and Corresponding Eigenvectors of Large Real-Symmetric Matrices. *Journal of Computational Physics*, 17:87–94, 1975.
19. W. W. Bradbury and R. Fletcher. New Iterative Methods for Solution of the Eigenproblem. *Numerische Mathematik*, 9:259–267, 1966.
20. A. Edelman and S. T. Smith. On conjugate gradient-like methods for eigenvalue-like problems. *BIT*, 36(3):494–508, 1996.
21. J. M. Badía and A. M. Vidal. Exploiting the Parallel Divide-and-Conquer Method to Solve the Symmetric Tridiagonal Eigenproblem. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing. Madrid, January 21-23*, 1998.
22. D. Giménez, V. Hernández and A. M. Vidal. A Unified Approach to Parallel Block-Jacobi Methods for the Symmetric Eigenvalue Problem. In *Proceedings of VECPAR '98*, 1998.

This article was processed using the LaTeX macro package with LLNCS style

Parallel Domain-Decomposition Preconditioning for Computational Fluid Dynamics

Timothy J. Barth[1], Tony F. Chan[2*], and Wei-Pai Tang[3**]

[1] NASA Ames Research Center, Mail Stop T27A-1,Moffett Field, CA 94035, USA
barth@nas.nasa.gov
[2] UCLA Department of Mathematics, Los Angeles, CA 90095-1555, USA
chan@math.ucla.edu
[3] University of Waterloo Department of Computer Science, Waterloo, Ontario N2L 3G1, Canada wptang@bz.uwaterloo.ca

Abstract. Algebraic preconditioning algorithms suitable for computational fluid dynamics (CFD) based on overlapping and non-overlapping domain decomposition (DD) are considered. Specific distinction is given to techniques well-suited for time-dependent and steady-state computations of fluid flow. For time-dependent flow calculations, the overlapping Schwarz algorithm suggested by Wu et al. [28] together with stabilized (upwind) spatial discretization shows excellent scalability and parallel performance without requiring a coarse space correction. For steady-state flow computations, a family of non-overlapping Schur complement DD techniques are developed. In the Schur complement DD technique, the triangulation is first partitioned into a number of non-overlapping subdomains and interfaces. A permutation of the mesh vertices based on subdomains and interfaces induces a natural 2×2 block partitioning of the discretization matrix. Exact LU factorization of this block system introduces a Schur complement matrix which couples subdomains and the interface together. A family of simplifying techniques for constructing the Schur complement and applying the 2×2 block system as a DD preconditioner are developed. Sample fluid flow calculations are presented to demonstrate performance characteristics of the simplified preconditioners.

1 Overview

The efficient numerical simulation of compressible fluid flow about complex geometries continues to be a challenging problem in large scale computing. Many

* The second author was partially supported by the National Science Foundation grant ASC-9720257, by NASA under contract NAS 2-96027 between NASA and the Universities Space Research Association (USRA).
** The third author was partially supported by NASA under contract NAS 2-96027 between NASA and the Universities Space Research Association (USRA), by a Natural Sciences and Engineering Research Council of Canada and by the Information Technology Research Centre which is funded by the Province of Ontario.

computational problems of interest in combustion, turbulence, aerodynamic performance analysis and optimization will require orders of magnitude increases in mesh resolution and/or solution degrees of freedom (dofs) to adequately resolve relevant fluid flow features. In solving these large problems, issues such as algorithmic scalability [1] and efficiency become fundamentally important. Furthermore, current computer hardware projections suggest that the needed computational resources can only be achieved via parallel computing architectures. Under this scenario, two algorithmic solution strategies hold particular promise in computational fluid dynamics (CFD) in terms of complexity and implementation on parallel computers: (1) multigrid (MG) and (2) domain decomposition (DD). Both are known to possess essentially optimal solution complexity for model discretized elliptic equations. Algorithms such as DD are particularly well-suited to distributed memory parallel computing architectures with high off-processor memory latency since these algorithms maintain a high degree of on-processor data locality. Unfortunately, it remains an open challenge to obtain similar optimal complexity results using DD and/or MG algorithms for the hyperbolic-elliptic and hyperbolic-parabolic equations modeling compressible fluid flow. In the remainder of this paper, we report on promising domain decomposition strategies suitable for the equations of CFD. In doing so, it is important to distinguish between two types of flow calculations:

1. *Steady-state computation of fluid flow.* The spatially hyperbolic-elliptic nature of the equations places special requirements on the solution algorithm. In the elliptic-dominated limit, global propagation of decaying error information is needed for optimality. This is usually achieved using either a coarse space operator (multigrid and overlapping DD methods) or a global interface operator (non-overlapping DD methods). In the hyperbolic-dominated limit, error components are propagated along characteristics of the flow. This suggests specialized coarse space operators (MG and overlapping DD methods) or special interface operators (non-overlapping DD methods). In later sections, both overlapping and non-overlapping DD methods are considered in further detail.
2. *Time-dependent computation of fluid flow.* The hyperbolic-parabolic nature of the equations is more forgiving. Observe that the introduction of numerical time integration implies that error information can only propagate over relatively small distances during a given time step interval. In the context of overlapping DD methods with backward Euler time integration, it becomes mathematically possible to show that scalability is retained without a coarse space correction by choosing the time step small enough and the subdomain overlap sufficiently large enough, cf. Wu et al. [28]. Section 5.3 reviews the relevant theory and examines the practical merits by performing time-dependent Euler equation computations using overlapping DD with no coarse space correction.

[1] the arithmetic complexity of algorithms with increasing number of degrees of freedom

2 Scalability and Preconditioning

To understand algorithmic scalability and the role of preconditioning, we think of the partial differential equation (PDE) discretization process as producing linear or linearized systems of equations of the form

$$Ax - b = 0 \tag{1}$$

where A is some large (usually sparse) matrix, b is a given right-hand-side vector, and x is the desired solution. For many practical problems, the amount of arithmetic computation required to solve (1) by iterative methods can be estimated in terms of the condition number of the system $\kappa(A)$. If A is symmetric positive definite (SPD), the well-known conjugate gradient method converges at a constant rate which depends on κ. After n iterations of the conjugate gradient method, the error ϵ satisfies

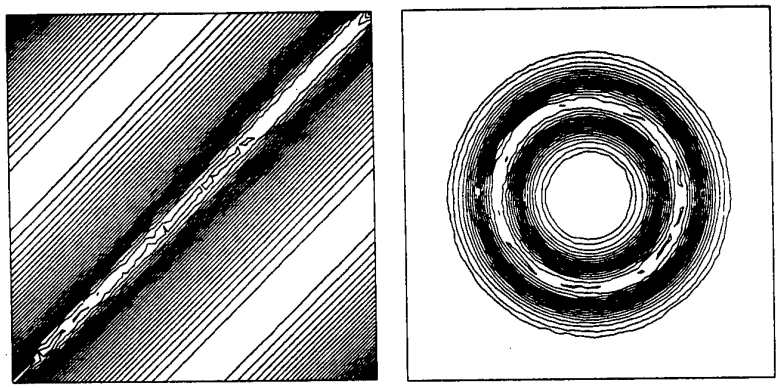
$$\frac{\|\epsilon^n\|_2}{\|\epsilon^0\|_2} \leq \left(\frac{\sqrt{\kappa(A)-1}}{\sqrt{\kappa(A)+1}} \right)^n . \tag{2}$$

For most applications of interest in computational fluid dynamics, the condition number associated with A depends on computational parameters such as the mesh spacing h, added stabilization terms, and/or artificial viscosity coefficients. In addition, $\kappa(A)$ can depend on physical parameters such as the Peclet number and flow direction as well as the underlying stability and well-posedness of the PDE and boundary conditions. Of particular interest in algorithm design and implementation is the parallel scalability experiment whereby a mesh discretization of the PDE is successively refined while keeping a fixed physical domain so that the mesh spacing h uniformly approaches zero. In this setting, the matrix A usually becomes increasingly ill-conditioned because of the dependence of $\kappa(A)$ on h. A standard technique to overcome this ill-conditioning is to solve the prototype linear system in right (or left) preconditioned form

$$(AP^{-1})Px - b = 0 \ . \tag{3}$$

The solution is unchanged but the convergence rate of iterative methods now depends on properties of AP^{-1}. Ideally, one seeks preconditioning matrices P which are easily solved and in some sense nearby A, e.g. $\kappa(AP^{-1}) = O(1)$ when A is SPD. The situation changes considerably for advection dominated problems. The matrix A ceases to be SPD so that the performance of iterative methods is not directly linked to the condition number behavior of A. Moreover, the convergence properties associated with A can depend on nonlocal properties of the PDE. To see this, consider the advection and advection-diffusion problems shown in Fig. 1. The entrance/exit flow shown in Fig. 1(a) transports the solution and any error components along 45° characteristics which eventually exit the domain. This is contrasted with the recirculation flow shown in Fig. 1(b) which has circular characteristics in the advection dominated limit. In this (singular) limit, any radially symmetric error components persist for all time. More generally, these

recirculation error components are removed by the physical *cross-wind* diffusion terms present in the PDE or the artificial cross-wind diffusion terms introduced by the numerical discretization. When the advection speed is large and the cross-wind diffusion small, the problem becomes ill-conditioned. Brandt and Yavneh [6] have studied both entrance/exit and recirculation flow within the context of multigrid acceleration. The behavior of multigrid (or most other iterative meth-



(a) 45° error component transport for entrance/exit flow.

(b) Radially symmetric error component for recirculating flow.

Fig. 1. Two model advection flows: (a) entrance/exit flow $u_x + u_y = 0$, (b) recirculating flow $yu_x - xu_y = \lim_{\epsilon \downarrow 0} \epsilon \Delta u$.

ods) for these two flow problems is notably different. For example, Fig. 2 graphs the convergence history of ILU(0)-preconditioned GMRES in solving Cuthill-McKee ordered matrix problems for entrance/exit flow and recirculation flow discretized using the Galerkin least-squares (GLS) procedure described in Sect. 3. The entrance/exit flow matrix problem is solved to a 10^{-8} accuracy tolerance in approximately 20 ILU(0)-GMRES iterations. The recirculation flow problem with $\epsilon = 10^{-3}$ requires 45 ILU(0)-GMRES iterations to reach the 10^{-8} tolerance and approximately 100 ILU(0)-GMRES iterations with $\epsilon = 0$. This difference in the number of iterations required for each problem increases dramatically as the mesh is refined. Using the non-overlapping DD method described in Sect. 5.5, we can remove the ill-conditioning observed in the recirculating flow problem. Let V_H denote the set of vertices along a nearly horizontal line from the center of the domain to the right boundary and V_S the set of remaining vertices in the mesh, see Fig. 3. Next, permute the discretization matrix so that solution unknowns corresponding to V_H are ordered last. The remaining mesh vertices have a natural ordering along characteristics of the advection operator which renders the discretization matrix associated with V_S nearly lower triangular. Using the technique of Sect. 5.5 together with exact factorization of the small $|V_H| \times |V_H|$

Schur complement, acceptable convergence rates for ILU(0)-preconditioned GMRES are once again obtainable as shown in Fig. 3. These promising results have strenghthened our keen interest in DD for fluid flow problems.

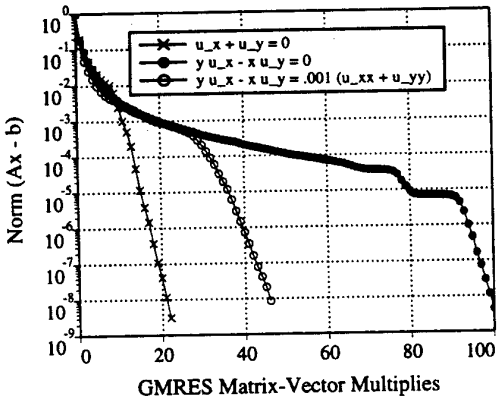
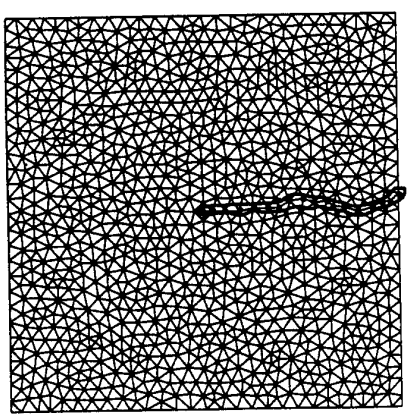
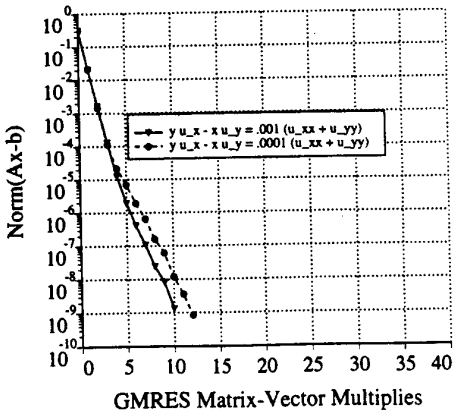


Fig. 2. Convergence behavior of ILU(0)-preconditioned GMRES for entrance/exit and recirculation flow problems using GLS discretization in a triangulated square (1600 dofs).



(a) Horizontal vertex set ordered last in matrix (circled vertices).



(b) ILU-GMRES convergence history.

Fig. 3. Sample mesh and ILU-GMRES convergence history using the non-overlapping DD technique of Sect. 5.5.

3 Stabilized Numerical Discretization

Non-overlapping domain-decomposition procedures such as those developed in Sect. 5.5 strongly motivate the use of compact-stencil spatial discretizations since larger discretization stencils produce larger interface sizes. For this reason, the Petrov-Galerkin approximation due to Hughes, Franca and Mallet [17,18] has been used in the present study. Consider the prototype conservation law system in m coupled independent variables in the spatial domain $\Omega \subset \mathbb{R}^d$ with boundary surface Γ and exterior normal $\mathbf{n}(x)$

$$\mathbf{u}_{,t} + \mathbf{f}^i_{,x_i} = 0, \quad (x,t) \in \Omega \times [0, \mathbb{R}^+] \tag{4}$$

$$(n_i \mathbf{f}^i_{,\mathbf{u}})^- (\mathbf{u} - \mathbf{g}) = 0, \quad (x,t) \in \Gamma \times [0, \mathbb{R}^+] \tag{5}$$

with implied summation over repeated indices. In this equation, $\mathbf{u} \in \mathbb{R}^m$ denotes the vector of conserved variables and $\mathbf{f}^i \in \mathbb{R}^m$ the inviscid flux vectors. The vector $\mathbf{g}$ can be suitably chosen to impose characteristic data or surface flow tangency using reflection principles. The conservation law system (4) is assumed to possess a generalized entropy pair so that the change of variables $\mathbf{u}(\mathbf{v}) : \mathbb{R}^m \mapsto \mathbb{R}^m$ symmetrizes the system in quasi-linear form

$$\mathbf{u}_{,\mathbf{v}} \mathbf{v}_{,t} + \mathbf{f}^i_{,\mathbf{v}} \mathbf{v}_{,x_i} = 0 \tag{6}$$

with $\mathbf{u}_{,\mathbf{v}}$ symmetric positive definite and $\mathbf{f}^i_{,\mathbf{v}}$ symmetric. The computational domain Ω is composed of non-overlapping simplicial elements T_i, $\Omega = \cup T_i$, $T_i \cap T_j = \emptyset$, $i \neq j$. For purposes of the present study, our attention is restricted to steady-state calculations. Time derivatives are retained in the Galerkin integral so that a pseudo-time marching strategy can be used for obtaining steady-state solutions. The Galerkin least-squares method due to Hughes, Franca and Mallet [17] can be defined via the following variational problem with time derivatives omitted from the least-squares bilinear form: Let $\mathcal{V}^h$ denote the finite element space $\mathcal{V}^h = \left\{ \mathbf{w}^h | \mathbf{w}^h \in \left(C^0(\Omega) \right)^m, \mathbf{w}^h|_T \in \left(\mathcal{P}_k(T) \right)^m \right\}$.

Find $\mathbf{v}^h \in \mathcal{V}^h$ such that for all $\mathbf{w}^h \in \mathcal{V}^h$

$$B(\mathbf{v}^h, \mathbf{w}^h)_{gal} + B(\mathbf{v}^h, \mathbf{w}^h)_{ls} + B(\mathbf{v}^h, \mathbf{w}^h)_{bc} = 0 \tag{7}$$

with

$$B(\mathbf{v}, \mathbf{w})_{gal} = \int_\Omega (\mathbf{w}^T \mathbf{u}(\mathbf{v})_{,t} - \mathbf{w}^T_{,x_i} \mathbf{f}^i(\mathbf{v}))\, d\Omega$$

$$B(\mathbf{v}, \mathbf{w})_{ls} = \sum_{T \in \Omega} \int_T \left(\mathbf{f}^i_{,\mathbf{v}} \mathbf{w}_{,x_i} \right)^T \tau \left(\mathbf{f}^i_{,\mathbf{v}} \mathbf{v}_{,x_i} \right) d\Omega$$

$$B(\mathbf{v}, \mathbf{w})_{bc} = \int_\Gamma \mathbf{w}^T \, \mathbf{h}(\mathbf{v}, \mathbf{g}; \mathbf{n}) \, d\Gamma$$

where

$$\mathbf{h}(\mathbf{v}_-, \mathbf{v}_+, \mathbf{n}) = \frac{1}{2} \left(\mathbf{f}(\mathbf{u}(\mathbf{v}_-); \mathbf{n}) + \mathbf{f}(\mathbf{u}(\mathbf{v}_+); \mathbf{n}) \right) - \frac{1}{2} |A(\mathbf{u}(\overline{\mathbf{v}}); \mathbf{n})| (\mathbf{u}(\mathbf{v}_+) - \mathbf{u}(\mathbf{v}_-)).$$

Inserting standard C^0 polynomial spatial approximations and mass-lumping of the remaining time derivative terms, yields coupled ordinary differential equations of the form:

$$D\,\mathbf{u}_t = \mathcal{R}(\mathbf{u}), \quad \mathcal{R}(\mathbf{u}) : \mathbb{R}^n \rightarrow \mathbb{R}^n \tag{8}$$

or in symmetric variables

$$D\,\mathbf{u}_{,\mathbf{v}}\mathbf{v}_t = \mathcal{R}(\mathbf{u}(\mathbf{v})) \tag{9}$$

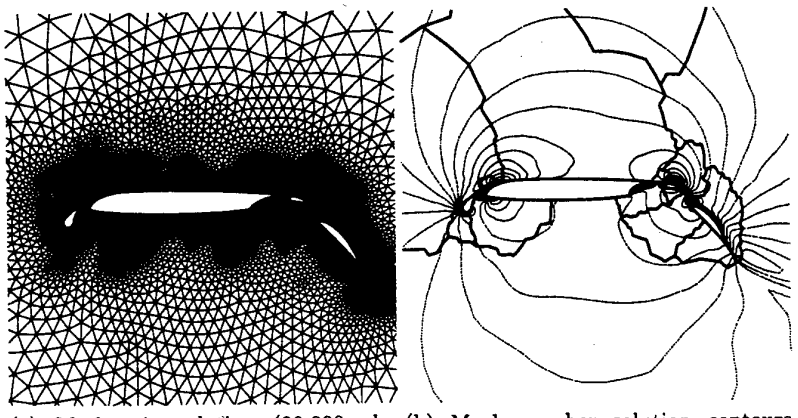
where D represents the (diagonal) lumped mass matrix. In the present study, backward Euler time integration with local time linearization is applied to Eqn. (8) yielding:

$$\left[\frac{1}{\Delta t}D - \left(\frac{\partial \mathcal{R}}{\partial \mathbf{u}}\right)^n\right](\mathbf{u}^{n+1} - \mathbf{u}^n) = \mathcal{R}(\mathbf{u}^n)\ . \tag{10}$$

The above equation can also be viewed as a modified Newton method for solving the steady-state equation $\mathcal{R}(\mathbf{u}) = 0$. For each modified Newton step, a large Jacobian matrix must be solved. In practice Δt is varied as an exponential function $\|\mathcal{R}(\mathbf{u})\|$ so that Newton's method is approached as $\|\mathcal{R}(\mathbf{u})\| \rightarrow 0$. Since each Newton iterate in (10) produces a linear system of the form (1), our attention focuses on this prototype linear form.

4 Domain Partitioning

In the present study, meshes are partitioned using the multilevel k-way partitioning algorithm METIS developed by Karypis and Kumar [19]. Figure 4(a) shows a typical airfoil geometry and triangulated domain. To construct a non-



(a) Mesh triangulation (80,000 elements).

(b) Mach number solution contours and partition (bold lines).

Fig. 4. Multiple component airfoil geometry with 16 subdomain partitioning and sample solution contours ($M_\infty = .20, \alpha = 10°$).

overlapping partitioning, a dual triangulation graph has been provided to the METIS partitioning software. Figure 4(b) shows partition boundaries and sample solution contours using the spatial discretization technique described in the previous section. By partitioning the dual graph of the triangulation, the number of elements in each subdomain is automatically balanced by the METIS software. Unfortunately, a large percentage of computation in our domain-decomposition algorithm is proportional to the interface size associated with each subdomain. On general meshes containing non-uniform element densities, balancing subdomain sizes does not imply a balance of interface sizes. In fact, results shown in Sect. 6 show increased imbalance of interface sizes as meshes are partitioned into larger numbers of subdomains. This ultimately leads to poor load balancing of the parallel computation. This topic will be revisited in Sect. 6.

5 Preconditioning Algorithms for CFD

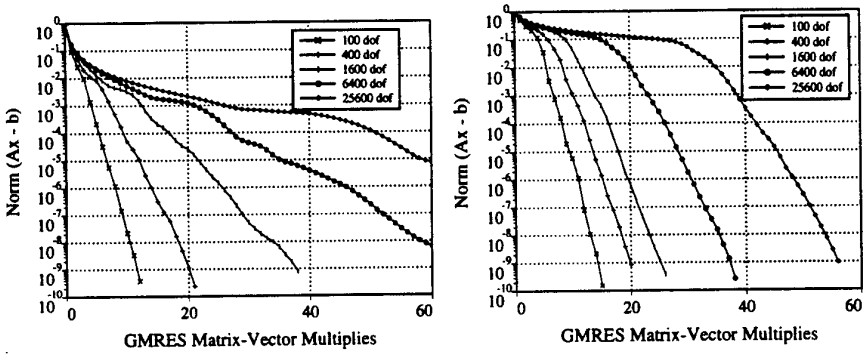
In this section, we consider several candidate preconditioning techniques based on overlapping and non-overlapping domain decomposition.

5.1 ILU Factorization

A common preconditioning choice is incomplete lower-upper factorization with arbitrary fill level k, ILU[k]. Early application and analysis of ILU preconditioning is given in Evans [15], Stone [27] and Meijerink and van der Vorst [21]. Although the technique is algebraic and well-suited to sparse matrices, ILU-preconditioned systems are not generally scalable. For example, Dupont et al. [14] have shown that ILU[0] preconditioning does not asymptotically change the $O(h^{-2})$ condition number of the 5-point difference approximation to Laplace's equation. Figure 5 shows the convergence of ILU-preconditioned GMRES for Cuthill-McKee ordered matrix problems obtained from diffusion and advection dominated problems discretized using Galerkin and Galerkin least-squares techniques respectively with linear elements. Both problems show pronounced convergence deterioration as the number of solution unknowns (degrees of freedom) increases. Note that matrix orderings exist for discretized scalar advection equations that are vastly superior to Cuthill-McKee ordering. Unfortunately, these orderings do not generalize naturally to coupled systems of equations which do not have a single characteristic direction. Some ILU matrix ordering experiments are given in [10]. Keep in mind that ILU *does* recluster eigenvalues of the preconditioned matrix so that for small enough problems a noticeable improvement can often be observed when ILU preconditioning is combined with a Krylov projection sequence.

5.2 Additive Overlapping Schwarz Methods

Let V denote the triangulation vertex set. Assume the triangulation has been partitioned into N overlapping subdomains with vertex sets $V_i, i = 1, \ldots, N$ such



(a) Diffusion dominated problem. (b) Advection dominated problem.

Fig. 5. Convergence dependence of ILU on the number of mesh points for diffusion and advection dominated problems using SUPG discretization and Cuthill-McKee ordering.

that

$$V = \cup_{i=1}^{N} V_i \ .$$

Let R_i denote the rectangular restriction matrix that returns the vector of coefficients in the subdomain Ω_i, i.e.

$$x_{\Omega_i} = R_i x \ .$$

Note that $A_i = R_i A R_i^T$ is the subdomain discretization matrix in Ω_i. The additive Schwarz preconditioner P^{-1} from (3) is then written as

$$P^{-1} = \sum_{i=1}^{N} R_i A_i^{-1} R_i^T \ .$$

The additive Schwarz algorithm [24] is appealing since each subdomain solve can be performed in parallel. Unfortunately the performance of the algorithm deteriorates as the number of subdomains increases. Let H denote the characteristic size of each subdomain, δ the overlap distance, and h the mesh spacing. Dryja and Widlund [12, 13] give the following condition number bound for the method when used as a preconditioner for elliptic discretizations

$$\kappa(AP^{-1}) \leq CH^{-2}\left(1 + (H/\delta)^2\right) \tag{11}$$

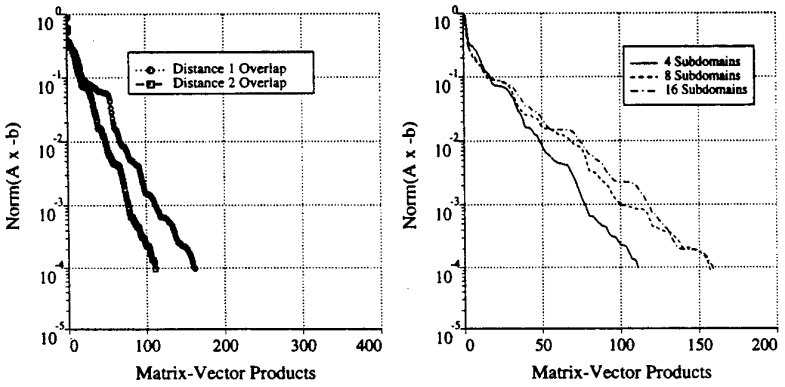
where C is a constant independent of H and h. This result describes the deterioration as the number of subdomains increases (and H decreases). With some additional work this deterioration can be removed by the introduction of a global coarse subspace with restriction matrix R_0 with scale H so that

$$P^{-1} = R_0 A R_0^T + \sum_{i=1}^{N} R_i A_i^{-1} R_i^T \ .$$

Under the assumption of "generous overlap" the condition number bound [12, 13, 8] can be improved to

$$\kappa(AP^{-1}) \leq C\,(1 + (H/\delta)) \quad . \tag{12}$$

The addition of a coarse space approximation introduces implementation problems similar to those found in multigrid methods described below. Once again,



(a) Effect of increasing mesh overlap. (b) Effect of increasing number of subdomains.

Fig. 6. Performance of GMRES with additive overlapping Schwarz preconditioning.

the theory associated with additive Schwarz methods for hyperbolic PDE systems is not well-developed. Practical applications of the additive Schwarz method for the steady-state calculation of hyperbolic PDE systems show similar deterioration of the method when the coarse space is omitted. Figure 6 shows the performance of the additive Schwarz algorithm used as a preconditioner for GMRES. The test matrix was taken from one step of Newton's method applied to an upwind finite volume discretization of the Euler equations at low Mach number ($M_\infty = .2$), see Barth [1] for further details. These calculations were performed without coarse mesh correction. As expected, the graphs show a degradation in quality with decreasing overlap and increasing number of mesh partitions.

5.3 Additive Overlapping Schwarz Methods for Time-Dependent Fluid Flow

We begin by giving a brief sketch of the analysis given in Wu et al. [28] which shows that for small enough time step and large enough overlap, the additive Schwarz preconditioner for hyperbolic problems behaves optimally without requiring a coarse space correction.

Consider the model scalar hyperbolic equation for the spatial domain $\Omega \subset \mathbb{R}^d$ with characteristic boundary data g weakly imposed on Γ

$$\begin{aligned} &\frac{\partial u}{\partial t} + \beta \cdot \nabla u + c\,u = 0, \quad (x,t) \in \Omega \times [0,T] \\ &(\beta \cdot n(x))^- (u - g) = 0, \quad x \in \Gamma \end{aligned} \tag{13}$$

with $\beta \in \mathbb{R}^d$, $c > 0$, and suitable initial data. Suppose that backward Euler time integration is employed ($u^n(x) = u(x, n\,\Delta t)$), so that (13) can then be written as

$$\beta \cdot \nabla u^n + \left(c + (\Delta t)^{-1}\right) u^n = f$$

with $f = (\Delta t)^{-1} u^{n-1}$. Next solve this equation using Galerkin's method (dropping the superscript n): Find $u \in H^1(\Omega)$

$$(\beta \cdot \nabla u, v) + \left(c + (\Delta t)^{-1}\right)(u,v) = (f,v) + < u - g, v >_- \quad \forall v \in H^1(\Omega)$$

where $(u,v) = \int_\Omega u\,v\;dx$ and $< u,v >_\pm = \int_\Gamma u\,v(\beta \cdot n(x))^\pm\;dx$. Recall that Galerkin's method for linear advection is iso-energetic modulo boundary conditions so that the symmetric part of the bilinear form is simply

$$A(u,v) = \left(c + (\Delta t)^{-1}\right)(u,v) + \frac{1}{2}\left(< u,v >_+ - < u,v >_-\right)$$

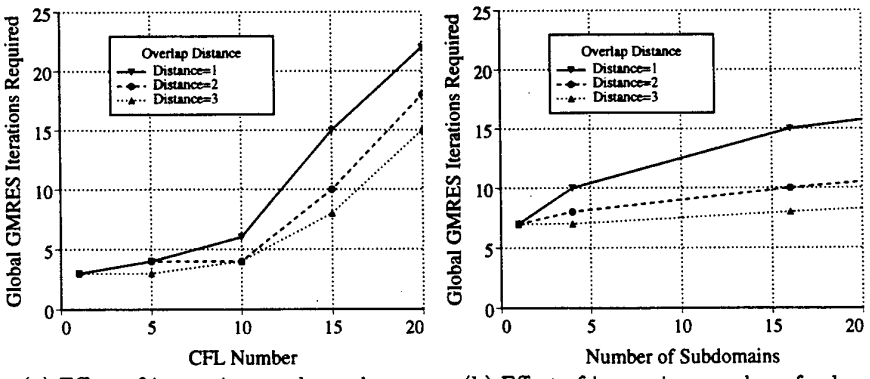
with skew-symmetric portion $S(u,v) = \frac{1}{2} < u,v > -(u, \beta \cdot \nabla v)$. Written in this form, it becomes clear that the term $\left(c + (\Delta t)^{-1}\right)(u,v)$ eventually dominates the skew-symmetric bilinear term if Δt is chosen small enough. This leads to the CFL-like assumption that $|\beta|\,\Delta t < h^{1+s}$, $s \geq 0$, see [28]. With this assumption, scalability of the overlapping Schwarz method without coarse space correction can be shown. Unfortunately, the assumed CFL-like restriction makes the method impractical since more efficient explicit time advancement strategies could be used which obviate the need for mesh overlap or implicit subdomain solves. The situation changes considerably if a Petrov-Galerkin discretization strategy if used such as described in Sect. 3. For a the scalar model equation (13) this amounts to added the symmetric bilinear stabilization term $B_{ls}(u,v) = (\beta \cdot \nabla u, \tau\beta \cdot \nabla v)$ to the previous Galerkin formulation: Find $u \in H^1(\Omega)$

$$(\beta \cdot \nabla u, v) + (\beta \cdot \nabla u, \tau\beta \cdot \nabla v) + \left(c + (\Delta t)^{-1}\right)(u,v) = (f,v) + < u - g, v >_- \quad \forall v \in H^1(\Omega)$$

where $\tau = h/(2|\beta|)$. This strategy is considered in a sequel paper to [28] which has yet to appear in the open literature. Practical CFD calculations show surprising good performance of overlapping Schwarz preconditioning when combined with Galerkin least-squares discretization of hyperbolic systems as discussed in Sect. 3. Figure 8 shows iso-density contours for Mach 3 flow over a backward-facing step geometry using a triangulated mesh containing 22000 mesh vertices which has been partitioned into 1, 4, 16, and 32 subdomains for evaluation purposes. Owing to the nonlinearity of the strong shockwave profiles, the solution

must be evolved in time at a relatively small Courant number < 20 to prevent nonlinear instability in the numerical method. On the other hand, the solution eventually reaches an equilibrium state. (Note that on finer resolution meshes, the fluid contact surface emanating from the Mach triple point eventually makes the flow field unsteady.) This test problem provides an ideal candidate scenario for the overlapping Schwarz method since time accuracy is not essential to reaching the correct steady-state solution. Computations were performed on a fixed

Fig. 7. Iso-density contours for Mach 3 inviscid Euler flow over a backward-facing step with exploded view of 16 subdomain partitioning.



(a) Effect of increasing mesh overlap.

(b) Effect of increasing number of subdomains ($CFL = 15$).

Fig. 8. Number of ILU(0)-GMRES iterations required to reduce $\|Ax - b\| < 10^{-5}$.

size mesh with 1, 4, 16, and 32 subdomains while also varying the overlap (graph) distances values and the CFL number. Figure 8(a) shows the effect of increasing

CFL number and subdomain mesh overlap distance on the number of global GMRES iterations required to solve the global matrix problem to an accuracy of less than 10^{-5} using additive Schwarz-like ILU(0) on overlapped subdomain meshes. For CFL numbers less than about 10, the number of GMRES iterations is relatively insensitive to the amount of overlap. Figure 8(b) shows the effect of increased mesh partitioning on the number of GMRES iterations required (assuming a fixed $CFL = 15$). For overlap distance ≥ 2, the iterative method is relatively insensitive to the number of subdomains. By lowering the CFL number to 10, the results become even less sensitive to the number of subdomains.

5.4 Multi-level Methods

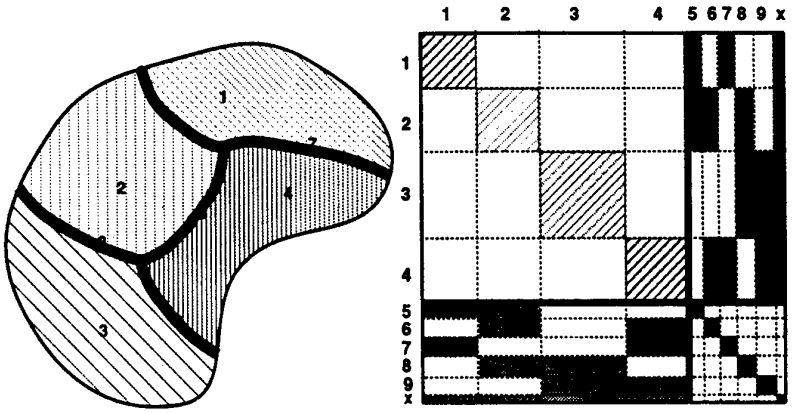
In the past decade, multi-level approaches such as multigrid has proven to be one of the most effective techniques for solving discretizations of elliptic PDEs [29]. For certain classes of elliptic problems, multigrid attains optimal scalability. For hyperbolic-elliptic problems such as the steady-state Navier-Stokes equations, the success of multigrid is less convincing. For example, Ref. [20] presents numerical results using multigrid to solve compressible Navier-Stokes flow about a multiple-component wing geometry with asymptotic convergence rates approaching .98 (Fig. 12 in Ref. [20]). This is quite far from the usual convergence rates quoted for multigrid on elliptic model problems. This is not too surprising since multigrid for hyperbolic-elliptic problems is not well-understood. In addition, some multigrid algorithms require operations such as mesh coarsening which are poorly defined for general meshes (especially in 3-D) or place unattainable shape-regularity demands on mesh generation. Other techniques add new meshing constraints to existing software packages which limit the overall applicability of the software. Despite the promising potential of multigrid for non-selfadjoint problems, we defer further consideration and refer the reader to works such as [6, 5].

5.5 Schur Complement Algorithms

Schur complement preconditioning algorithms are a general family of algebraic techniques in non-overlapping domain-decomposition. These techniques can be interpreted as variants of the well-known substructuring method introduced by Przemieniecki [22] in structural analysis. When recursively applied, the method is related to the nested dissection algorithm. In the present development, we consider an arbitrary domain as illustrated in Fig. 9 that has been further decomposed into subdomains labeled $1-4$, interfaces labeled $5-9$, and cross points x. A natural 2×2 partitioning of the system is induced by permuting rows and columns of the discretization matrix so that subdomain unknowns are ordered first, interface unknowns second, and cross points ordered last

$$Ax = \begin{bmatrix} A_{\mathcal{D}\mathcal{D}} & A_{\mathcal{D}\mathcal{I}} \\ A_{\mathcal{I}\mathcal{D}} & A_{\mathcal{I}\mathcal{I}} \end{bmatrix} \begin{pmatrix} x_{\mathcal{D}} \\ x_{\mathcal{I}} \end{pmatrix} = \begin{pmatrix} f_{\mathcal{D}} \\ f_{\mathcal{I}} \end{pmatrix} \tag{14}$$

where $x_{\mathcal{D}}, x_{\mathcal{I}}$ denote the subdomain and interface variables respectively. The



(a) Partitioned domain.

(b) Induced 2 × 2 block discretization matrix.

Fig. 9. Domain decomposition and the corresponding block matrix.

block LU factorization of A is then given by

$$A = LU = \begin{bmatrix} A_{\mathcal{D}\mathcal{D}} & 0 \\ A_{\mathcal{I}\mathcal{D}} & I \end{bmatrix} \begin{bmatrix} I & A_{\mathcal{D}\mathcal{D}}^{-1} A_{\mathcal{D}\mathcal{I}} \\ 0 & S \end{bmatrix}, \tag{15}$$

where

$$S = A_{\mathcal{I}\mathcal{I}} - A_{\mathcal{I}\mathcal{D}} A_{\mathcal{D}\mathcal{D}}^{-1} A_{\mathcal{D}\mathcal{I}} \tag{16}$$

is the Schur complement for the system. Note that $A_{\mathcal{D}\mathcal{D}}$ is block diagonal with each block associated with a subdomain matrix. Subdomains are decoupled from each other and only coupled to the interface. The subdomain decoupling property is exploited heavily in parallel implementations.

In the next section, we outline a naive parallel implementation of the "exact" factorization. This will serve as the basis for a number of simplifying approximations that will be discussed in later sections.

5.6 "Exact" Factorization

Given the domain partitioning illustrated in Fig. 9, a straightforward (but naive) parallel implementation would assign a processor to each subdomain and a single processor to the Schur complement. Let $\overline{\mathcal{I}}_i$ denote the union of interfaces surrounding $\mathcal{D}_i$. The entire solution process would then consist of the following steps:

Parallel Preprocessing:

1. Parallel computation of subdomain $A_{\mathcal{D}_i\mathcal{D}_i}$ matrix LU factors.

2. Parallel computation of Schur complement block entries associated with each subdomain $\mathcal{D}_i$

$$\Delta S_{\overline{\mathcal{I}}_i} = A_{\overline{\mathcal{I}}_i \mathcal{D}_i} A^{-1}_{\mathcal{D}_i \mathcal{D}_i} A_{\mathcal{D}_i \overline{\mathcal{I}}_i} \ . \tag{17}$$

3. Accumulation of the global Schur complement S matrix

$$S = A_{\mathcal{I}\mathcal{I}} - \sum_{i=1}^{\#subdomains} \Delta S_{\overline{\mathcal{I}}_i} \ . \tag{18}$$

Solution:

Step (1) $u_{\mathcal{D}_i} = A^{-1}_{\mathcal{D}_i \mathcal{D}_i} b_{\mathcal{D}_i}$	(parallel)
Step (2) $v_{\overline{\mathcal{I}}_i} = A_{\overline{\mathcal{I}}_i \mathcal{D}_i} u_{\mathcal{D}_i}$	(parallel)
Step (3) $w_{\mathcal{I}} = b_{\mathcal{I}} - \sum_{i=1}^{\#subdomains} v_{\overline{\mathcal{I}}_i}$	(communication)
Step (4) $x_{\mathcal{I}} = S^{-1} w_{\mathcal{I}}$	(sequential, communication)
Step (5) $y_{\mathcal{D}_i} = A_{\mathcal{D}_i \overline{\mathcal{I}}_i} x_{\overline{\mathcal{I}}_i}$	(parallel)
Step (6) $x_{\mathcal{D}_i} = u_{\mathcal{D}_i} - A^{-1}_{\mathcal{D}_i \mathcal{D}_i} y_{\mathcal{D}_i}$	(parallel)

This algorithm has several deficiencies. Steps 3 and 4 of the solution process are sequential and require communication between the Schur complement and subdomains. More generally, the algorithm is not scalable since the growth in size of the Schur complement with increasing number of subdomains eventually overwhelms the calculation in terms of memory, computation, and communication.

5.7 Iterative Schur Complement Algorithms

A number of approximations have been investigated in Barth et al. [2] which simplify the exact factorization algorithm and address the growth in size of the Schur complement. During this investigation, our goal has been to develop algebraic techniques which can be applied to both elliptic and hyperbolic partial differential equations. These approximations include iterative (Krylov projection) subdomain and Schur complement solves, element dropping and other sparsity control strategies, localized subdomain solves in the formation of the Schur complement, and partitioning of the interface and parallel distribution of the Schur complement matrix. Before describing each approximation and technique, we can make several observations:

Observation 1. (Ill-conditioning of Subproblems) For model elliptic problem discretizations, it is known in the two subdomain case that $\kappa(A_{\mathcal{D}_i \mathcal{D}_i}) = O((L/h)^2)$ and $\kappa(S) = O(L/h)$ where L denotes the domain size. From this perspective, both subproblems are ill-conditioned since the condition number depends on the mesh spacing parameter h. If one considers the scalability experiment, the situation changes in a subtle way. In the scalability experiment, the number of mesh points and the number of subdomains is increased such that the ratio of subdomain size to mesh spacing size H/h is held constant. The

subdomain matrices for elliptic problem discretizations now exhibit a $O((H/h)^2)$ condition number so the cost associated with iteratively solving them (with or without preconditioning) is approximately constant as the problem size is increased. Therefore, this portion of the algorithm is scalable. Even so, it may be desirable to precondition the subdomain problems to reduce the overall cost. The Schur complement matrix retains (at best) the $O(L/h)$ condition number and becomes increasingly ill-conditioned as the mesh size is increased. Thus in the scalability experiment, it is ill-conditioning of the Schur complement matrix that must be controlled by adequate preconditioning, see for example Dryja, Smith and Widlund [11].

Observation 2. (Non-stationary Preconditioning) The use of Krylov projection methods to solve the local subdomain and Schur complement subproblems renders the global preconditioner non-stationary. Consequently, Krylov projection methods designed for non-stationary preconditioners should be used for the global problem. For this reason, FGMRES [23], a variant of GMRES designed for non-stationary preconditioning, has been used in the present work.

Observation 3. (Algebraic Coarse Space) The Schur complement serves as an algebraic coarse space operator since the system

$$Sx_{\mathcal{I}} = b_{\mathcal{I}} - A_{\mathcal{I}\mathcal{D}}A_{\mathcal{D}\mathcal{D}}^{-1}b_{\mathcal{D}} \tag{19}$$

globally couples solution unknowns on the entire interface. The rapid propagation of information to large distances is a crucial component of optimal algorithms.

5.8 ILU-GMRES Subdomain and Schur complement Solves

The first natural approximation is to replace exact inverses of the subdomain and Schur complement subproblems with an iterative Krylov projection method such as GMRES (or stabilized biconjugate gradient).

Iterative Subdomain Solves Recall from the exact factorization algorithm that a subdomain solve is required once in the preprocessing step and twice in the solution step. This suggests replacing these three inverses with m_1, m_2, and m_3 steps of GMRES respectively. As mentioned in Observation 1, although the condition number of subdomain problems remains roughly constant in the scalability experiment, it still is beneficial to precondition subdomain problems to improve the overall efficiency of the global preconditioner. By preconditioning subdomain problems, the parameters m_1, m_2, m_3 can be kept small. This will be exploited in later approximations. Since the subdomain matrices are assumed given, it is straightforward to precondition subdomains using ILU[k]. For the GLS spatial discretization, satisfactory performance is achieved using ILU[2].

Iterative Schur complement Solves It is possible to avoid explicitly computing the Schur complement matrix for use in Krylov projection methods by alternatively computing the action of S on a given vector p, i.e.

$$S\,p = A_{\mathcal{I}\mathcal{I}}\,p - A_{\mathcal{I}\mathcal{D}}A_{\mathcal{D}\mathcal{D}}^{-1}A_{\mathcal{D}\mathcal{I}}\,p \ . \tag{20}$$

Unfortunately S is ill-conditioned, thus some form of interface preconditioning is needed. For elliptic problems, the rapid decay of elements away from the diagonal in the Schur complement matrix [16] permits simple preconditioning techniques. Bramble, Pasciak, and Schatz [4] have shown that even the simple block Jacobi preconditioner yields a substantial improvement in condition number

$$\kappa(SP_S^{-1}) \leq CH^{-2}\left(1+\log^2(H/h)\right) \tag{21}$$

for C independent of h and H. For a small number of subdomains, this technique is very effective. To avoid the explicit formation of the diagonal blocks, a number of simplified approximations have been introduced over the last several years, see for examples Bjorstad [3] or Smith et al. [26]. By introducing a further coarse space coupling of cross points to the interface, the condition number is further improved

$$\kappa(SP_S^{-1}) \leq C\left(1+\log^2(H/h)\right) \ . \tag{22}$$

Unfortunately, the Schur complement associated with advection dominated discretizations may not exhibit the rapid element decay found in the elliptic case. This can occur when characteristic trajectories of the advection equation traverse a subdomain from one interface edge to another. Consequently, the Schur complement is not well-preconditioned by elliptic-like preconditioners that use the action of local problems. A more basic strategy has been developed in the present work whereby elements of the Schur complement are *explicitly computed.* Once the elements have been computed, ILU factorization is used to precondition the Schur complement iterative solution. In principle, ILU factorization with a suitable reordering of unknowns can compute the long distance interactions associated with simple advection fields corresponding to entrance/exit-like flows. For general advection fields, it remains a topic of current research to find reordering algorithms suitable for ILU factorization. The situation is further complicated for coupled systems of hyperbolic equations (even in two independent variables) where multiple characteristic directions and/or Cauchy-Riemann systems can be produced. At the present time, Cuthill-McKee ordering has been used on all matrices although improved reordering algorithms are currently under development.

In the present implementation, each subdomain processor computes (in parallel) and stores portions of the Schur complement matrix

$$\Delta S_{\overline{\mathcal{I}}_i} = A_{\overline{\mathcal{I}}_i\mathcal{D}_i}A_{\mathcal{D}_i\mathcal{D}_i}^{-1}A_{\mathcal{D}_i\overline{\mathcal{I}}_i} \ . \tag{23}$$

To gain improved parallel scalability, the interface edges and cross points are partitioned into a smaller number of generic "subinterfaces". This subinterface partitioning is accomplished by assigning a supernode to each interface edge

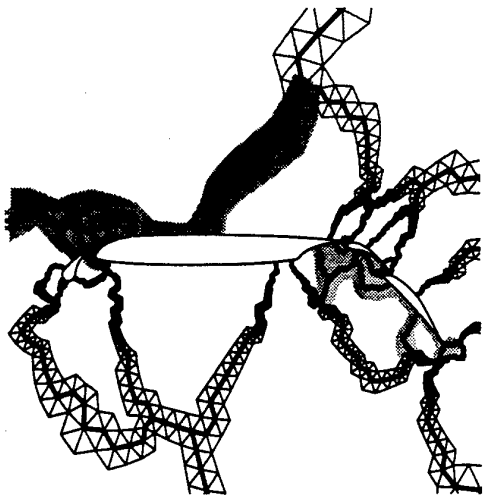


Fig. 10. Interface (bold lines) decomposed into 4 subinterfaces indicated by alternating shaded regions.

separating two subdomains, forming the graph of the Schur complement matrix in terms of these supernodes, and applying the METIS partitioning software to this graph. Let $\overline{\overline{\mathcal{I}}}_j$ denote the j-th subinterface such that $\mathcal{I} = \cup_j \overline{\overline{\mathcal{I}}}_j$. Computation of the action of the Schur complement matrix on a vector p needed in Schur complement solves now takes the (highly parallel) form

$$Sp = \sum_{j=1}^{\#subinterfaces} A_{\overline{\overline{\mathcal{I}}}_j \overline{\overline{\mathcal{I}}}_j} \, p(\overline{\overline{\mathcal{I}}}_j) - \sum_{i=1}^{\#subdomains} \Delta S_{\overline{\mathcal{I}}_i} \, p(\overline{\mathcal{I}}_i) \ . \tag{24}$$

Using this formula it is straightforward to compute the action of S on a vector p to any required accuracy by choosing the subdomain iteration parameter m_i large enough. Figure 10 shows an interface and the immediate neighboring mesh that has been decomposed into 4 smaller subinterface partitions for a 32 subdomain partitioning. By choosing the number of subinterface partitions proportional to the square root of the number of 2-D subdomains and assigning a processor to each, the number of solution unknowns associated with each subinterface is held approximately constant in the scalability experiment. Note that the use of iterative subdomain solves renders both Eqns. (20) and (24) approximate.

In our investigation, the Schur complement is preconditioned using ILU factorization. This is not a straightforward task for two reasons: (1) portions of the Schur complement are distributed among subdomain processors, (2) the interface itself has been distributed among several subinterface processors. In the next section, a block element dropping strategy is proposed for gathering por-

tions of the Schur complement together on subinterface processors for use in ILU preconditioning the Schur complement solve. Thus, a block Jacobi preconditioner is constructed for the Schur complement which is more powerful than the Bramble, Pasciak, and Schatz (BPS) form (without coarse space correction) since the blocks now correspond to larger subinterfaces rather than the smaller interface edges. Formally, BPS preconditioning without coarse space correction can be obtained for 2D elliptic discretizations by dropping additional terms in our Schur complement matrix approximation and ordering unknowns along interface edges so that the ILU factorization of the tridiagonal-like system for each interface edge becomes exact.

Block Element Dropping In our implementation, portions of the Schur complement residing on subdomain processors are gathered together on subinterface processors for use in ILU preconditioning of the Schur complement solve. In assembling a Schur complement matrix approximation on each subinterface processor, certain matrix elements are neglected:

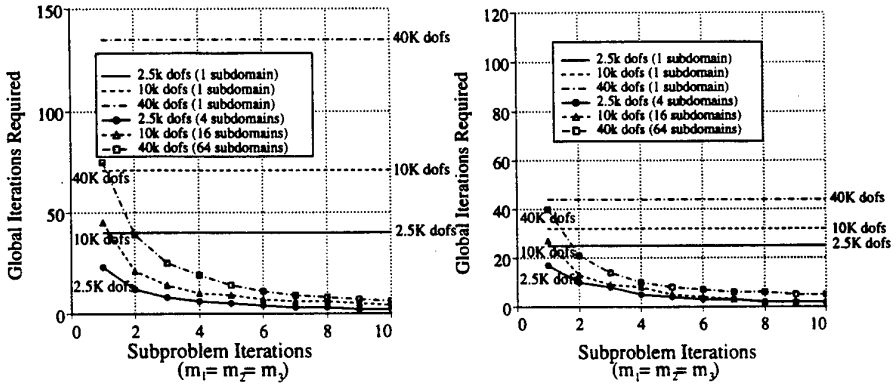
1. All elements that couple subinterfaces are ignored. This yields a block Jacobi approximation for subinterfaces.
2. All elements with matrix entry location that exceeds a user specified graph distance from the diagonal as measured on the triangulation graph are ignored. Recall that the Schur complement matrix can be very dense. The graph distance criteria is motivated by the rapid decay of elements away from the matrix diagonal for elliptic problems. In all subsequent calculations, a graph distance threshold of 2 has been chosen for block element dropping.

Figures 11(a) and 11(b) show calculations performed with the present non-overlapping domain-decomposition preconditioner for diffusion and advection problems. These figures graph the number of global FGMRES iterations needed to solve the discretization matrix problem to 10^{-6} accuracy tolerance as a function of the number of subproblem iterations. In this example, all the subproblem iteration parameters have been set equal to each other ($m_1 = m_2 = m_3$). The horizontal lines show poor scalability of single domain ILU-FGMRES on meshes containing 2500, 10000, and 40000 solution unknowns. The remaining curves show the behavior of the Schur complement preconditioned FGMRES on 4, 16, and 64 subdomain meshes. Satisfactory scalability for very small values (5 or 6) of the subproblem iteration parameter m_i is clearly observed.

Wireframe Approximation A major cost in the explicit construction of the Schur complement is the matrix-matrix product

$$A^{-1}_{\mathcal{D}_i\mathcal{D}_i} A_{\mathcal{D}_i\overline{\mathcal{I}}_i} \ . \tag{25}$$

Since the subdomain inverse is computed iteratively using ILU-GMRES iteration, forming (25) is equivalent to solving a multiple right-hand sides system with



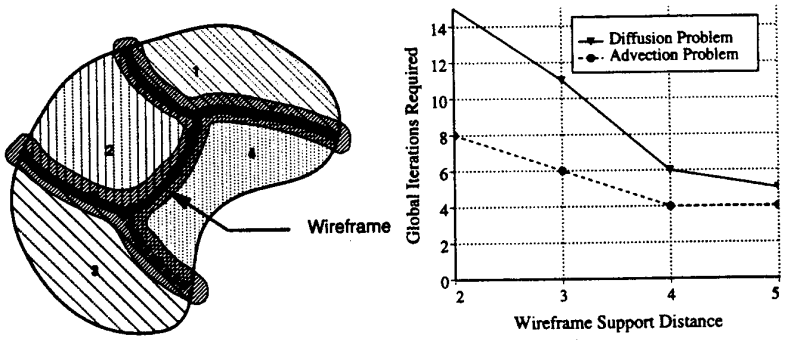
(a) Diffusion dominated problem. $u_{xx} + u_{yy} = 0$.

(b) Advection dominated problem. $u_x + u_y = 0$.

Fig. 11. Effect of the subproblem iteration parameters m_i on the global FGMRES convergence, $m_1 = m_2 = m_3$ for meshes containing 2500, 10000, and 40000 solution unknowns.

each right-hand side vector corresponding to a column of $A_{\mathcal{D}_i\overline{\mathcal{I}}_i}$. The number of columns of $A_{\mathcal{D}_i\overline{\mathcal{I}}_i}$ is precisely the number of solution unknowns located on the interface surrounding a subdomain. This computational cost can be quite large. Numerical experiments with Krylov projection methods designed for multiple right-hand side systems [25] showed only marginal improvement owing to the fact that the columns are essentially independent. In the following paragraphs, "wireframe" and "supersparse" approximations are introduced to reduce the cost in forming the Schur complement matrix.

The wireframe approximation idea [9] is motivated from standard elliptic domain-decomposition theory by the rapid decay of elements in S with graph distance from the diagonal. Consider constructing a relatively thin *wireframe* region surrounding the interface as shown in Fig. 12(a). In forming the Eqn. (25) expression, subdomain solves are performed using the much smaller wireframe subdomains. In matrix terms, a principal submatrix of A, corresponding to the variables within the wireframe, is used to compute the (approximate) Schur complement of the interface variables. It is known from domain-decomposition theory that the exact Schur complement of the wireframe region is spectrally equivalent to the Schur complement of the whole domain. This wireframe approximation leads to a substantial savings in the computation of the Schur complement matrix. Note that the full subdomain matrices are used everywhere else in the Schur complement algorithm. The wireframe technique introduces a new adjustable parameter into the preconditioner which represents the width of the wireframe. For simplicity, this width is specified in terms of graph distance on the mesh triangulation. Figure 12(b) demonstrates the performance of this approximation by graphing the total number of preconditioned FGMRES iterations



(a) Wireframe region surrounding interface.

(b) Effect of wireframe support on preconditioner performance for diffusion $u_{xx}+u_{yy}=0$ and advection $u_x+u_y=0$ problems.

Fig. 12. Wireframe region surrounding interface and preconditioner performance results for a fixed mesh size (1600 vertices) and 16 subdomain partitioning.

required to solve the global matrix problem to a 10^{-6} accuracy tolerance while varying the width of the wireframe. As expected, the quality of the preconditioner improves rapidly with increasing wireframe width with full subdomain-like results obtained using modest wireframe widths. As a consequence of the wireframe construction, the time taken form the Schur complement has dropped by approximately 50%.

Supersparse Matrix-Vector Operations It is possible to introduce further approximations which improve upon the overall efficiency in forming the Schur complement matrix. One simple idea is to exploit the extreme sparsity in columns of $A_{\mathcal{D}_i\overline{\mathcal{I}}_i}$ or equivalently the sparsity in the right-hand sides produced from $A^{-1}_{\mathcal{D}_i\mathcal{D}_i}A_{\mathcal{D}_i\overline{\mathcal{I}}_i}$ needed in the formation of the Schur complement. Observe that m steps of GMRES generates a small sequence of Krylov subspace vectors $[p, A\ p, A^2\ p, \ldots, A^m\ p]$ where p is a right-hand side vector. Consequently for small m, if both A and p are sparse then the sequence of matrix-vector products will be relatively sparse. Standard sparse matrix-vector product subroutines utilize the matrix in sparse storage format and the vector in dense storage format. In the present application, the vectors contain only a few non-zero entries so that standard sparse matrix-vector products waste many arithmetic operations. For this reason, a "supersparse" software library have been developed to take advantage of the sparsity in matrices as well as in vectors by storing both in compressed form. Unfortunately, when GMRES is preconditioned using ILU factorization, the Krylov sequence becomes $[p, A\,P^{-1}\ p, (A\,P^{-1})^2\ p, \ldots, (A\,P^{-1})^m\ p]$. Since the inverse of the ILU approximate factors $\widetilde{L}$ and $\widetilde{U}$ can be dense, the first application of ILU preconditioning produces a dense Krylov vector result. All subsequent

Krylov vectors can become dense as well. To prevent this densification of vectors using ILU preconditioning, a fill-level-like strategy has been incorporated into the ILU *backsolve* step. Consider the ILU preconditioning problem, $\widetilde{L}\widetilde{U}r = b$. This system is conventionally solved by a lower triangular backsolve, $w = \widetilde{L}^{-1}b$, followed by a upper triangular backsolve $r = \widetilde{U}^{-1}w$. In our supersparse strategy, sparsity is controlled by imposing a non-zero fill pattern for the vectors w and r during lower and upper backsolves. The backsolve fill patterns are most easily specified in terms fill-level distance, i.e. graph distance from existing nonzeros of the right-hand side vector in which new fill in the resultant vector is allowed to occur. This idea is motivated from the element decay phenomena observed for elliptic problems. Table 1 shows the performance benefits of using supersparse computations together with backsolve fill-level specification for a 2-D test problem consisting of Euler flow past a multi-element airfoil geometry partitioned into 4 subdomains with 1600 mesh vertices in each subdomain. Computations

Table 1. Performance of the Schur complement preconditioner with supersparse arithmetic for a 2-D test problem consisting of Euler flow past a multi-element airfoil geometry partitioned into 4 subdomains with 1600 mesh vertices in each subdomain.

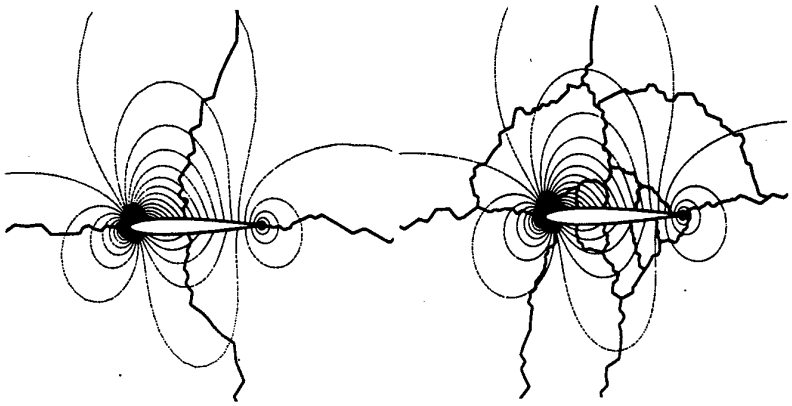
Backsolve Fill-Level Distance k	Global GMRES Iterations	Time(k)/Time(∞)
0	26	0.325
1	22	0.313
2	21	0.337
3	20	0.362
4	20	0.392
∞	20	1.000

were performed on the IBM SP2 parallel computer using MPI message passing protocol. Various values of backsolve fill-level distance were chosen while monitoring the number of global GMRES iterations needed to solve the matrix problem and the time taken to form the Schur complement preconditioner. Results for this problem indicate preconditioning performance comparable to exact ILU backsolves using backsolve fill-level distances of only 2 or 3 with a 60-70% reduction in cost.

6 Numerical Results on the IBM SP2

In the remaining paragraphs, we assess the performance of the Schur complement preconditioned FGMRES in solving linear matrix problems associated an approximate Newton method for the nonlinear discretized compressible Euler equations. All calculations were performed on an IBM SP2 parallel computer using MPI message passing protocol. A scalability experiment was per-

formed on meshes containing 4/1, 16/2, and 64/4 subdomains/subinterfaces with each subdomain containing 5000 mesh elements. Figures 13(a) and 13(b)



(a) Mach contours (4 subdomains, 20K elements). (b) Mach contours (16 subdomains, 80K elements).

Fig. 13. Mach number contours and mesh partition boundaries for NACA0012 airfoil geometry.

show mesh partitionings and sample Mach number solution contours for subsonic ($M_\infty = .20, \alpha = 2.0°$) flow over the airfoil geometry. The flow field was computed using the stabilized GLS discretization and approximate Newton method described in Sect. 3. Figure 14 graphs the convergence of the approximate Newton method for the 16 subdomain test problem. Each approximate Newton iterate shown in Fig. 14 requires the solution of a linear matrix system which has been solved using the Schur complement preconditioned FGMRES algorithm. Figure 15 graphs the convergence of the FGMRES algorithm for each matrix from the 4 and 16 subdomain test problems. These calculations were performed using ILU[2] and $m_1 = m_2 = m_3 = 5$ iterations on subproblems with supersparse distance equal to 5. The 4 subdomain mesh with 20000 total elements produces matrices that are easily solved in 9-17 global FGMRES iterations. Calculations corresponding to the largest CFL numbers are close approximations to exact Newton iterates. As is typically observed by these methods, the final few Newton iterates are solved more easily than matrices produced during earlier iterates. The most difficult matrix problem required 17 FGMRES iterations and the final Newton iterate required only 12 FGMRES iterations. The 16 subdomain mesh containing 80000 total elements produces matrices that are solved in 12-32 global FGMRES. Due to the nonlinearity in the spatial discretization, several approximate Newton iterates were relatively difficult to solve, requiring over 30 FGMRES iterations. As nonlinear convergence is obtained the matrix problems become less demanding. In this case, the final Newton iterate matrix

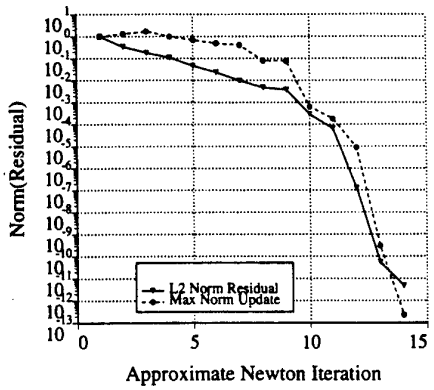
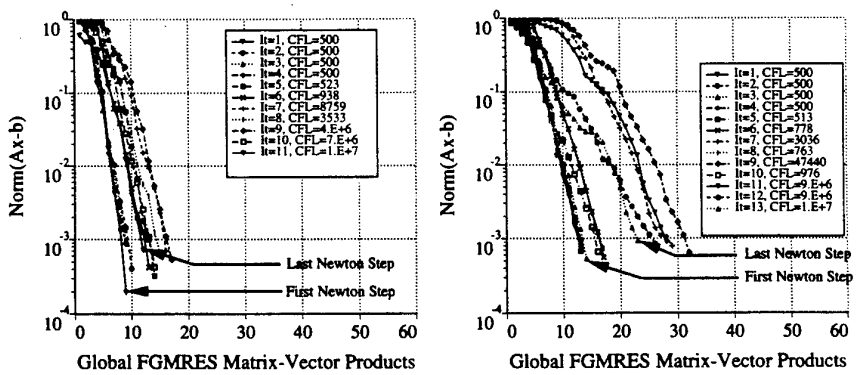


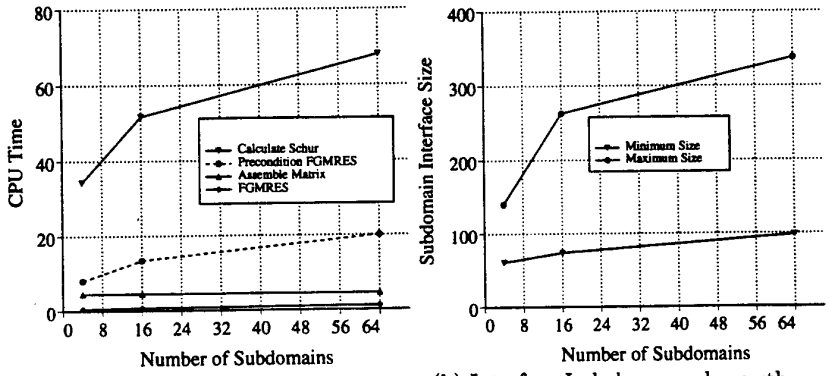
Fig. 14. Nonlinear convergence behavior of the approximate Newton method for subsonic airfoil flow.



(a) 4 subdomains (20K elements). (b) 16 subdomains (80K elements).

Fig. 15. FGMRES convergence history for each Newton step.

required 22 FGMRES iterations. This iteration degradation from the 4 subdomain case can be reduced by increasing the subproblem iteration parameters m_1, m_2, m_3 but the overall computation time is increased. In the remaining timing graphs, we have sampled timings from 15 FGMRES iterations taken from the final Newton iterate on each mesh. For example, Fig. 16(a) gives a raw timing breakdown for several of the major calculations in the overall solver: calculation of the Schur complement matrix, preconditioning FGMRES with the Schur complement algorithm, matrix element computation and assembly, and FGMRES solve. Results are plotted on each of the meshes containing 4, 16, and 64 subdomains with 5000 elements per subdomain. Since the number of elements in each subdomain is held constant, the time taken to assemble the matrix is



(a) Raw timing breakdown. (b) Interface Imbalance and growth.

Fig. 16. Raw IBM SP2 timing breakdown and the effect of increased number of subdomains on smallest and largest interface sizes.

also constant. Observe that in our implementation the time to form and apply the Schur complement preconditioner currently dominates the calculation. Although the growth observed in these timings with increasing numbers of subdomains comes from several sources, the dominate effect comes from a very simple source: ***the maximum interface size growth associated with subdomains.*** This has a devastating impact on the parallel performance since at the Schur complement synchronization point all processors must wait for subdomains working on the largest interfaces to finish. Figure 16(b) plots this growth in maximum interface size as a function of number of subdomains in our scalability experiment. Although the number of elements in each subdomain has been held constant in this experiment, the largest interface associated with any subdomain has more than doubled. This essentially translates into a doubling in time to form the Schur complement matrix. This doubling in time is clearly observed in the raw timing breakdown in Fig. 16(a). At this point in time, we known of no partitioning method that actively addresses controlling the maximum interface size associated with subdomains. We suspect that other non-overlapping methods are sensitive to this effect as well.

7 Concluding Remarks

Experience with our non-overlapping domain-decomposition method with an algebraically generated coarse problem shows that we can successfully trade off some of the robustness of the exact Schur complement method for increased efficiency by making appropriately designed approximations. In particular, the localized wireframe approximation and the supersparse matrix-vector operations together result in reduced cost without significantly degrading the overall convergence rate.

It remains an outstanding problem to partition domains such that the maximum interface size does grow with increased number of subdomains and mesh size. In addition, it may be cost effective to combine this technique with multigrid or multiple-grid techniques to improve the robustness of Newton's method.

References

1. T. J. Barth, *Parallel CFD algorithms on unstructured meshes*, Tech. Report AGARD Report R-907, Advisory Group for Aerospace Research and Development, 1995.
2. T. J. Barth, T. F. Chan, and W.-P. Tang, *A parallel non-overlapping domain-decomposition algorithm for compressible fluid flow problems on triangulated domains*, AMS Cont. Math. **218** (1998).
3. P. Bjorstad and O. B. Widlund, *Solving elliptic problems on regions partitioned into substructures*, SIAM J. Numer. Anal. **23** (1986), no. 6, 1093–1120.
4. J. H. Bramble, J. E. Pasciak, and A. H. Schatz, *The construction of preconditioners for elliptic problems by substructuring, I*, Math. Comp. **47** (1986), no. 6, 103–134.
5. J. H. Bramble, J. E. Pasciak, and J. Xu, *The analysis of multigrid algorithms for nonsymmetric and indefinite elliptic problems*, Math. Comp. **51** (1988), 289–414.
6. A. Brandt and I. Yavneh, *Accelerated multigrid convergence for high-Reynolds recirculating flow*, SIAM J. Sci. Comput. **14** (1993), 607–626.
7. X.-C. Cai, *Some domain decomposition algorithms for nonself-adjoint elliptic and parabolic partial differential equations*, Ph.D. thesis, Ph.D. Thesis, Courant Institute, 1989.
8. T. Chan and J. Zou, *Additive Schwarz domain decomposition methods for elliptic problems on unstructured meshes*, Tech. Report CAM 93-40, UCLA Department of Mathematics, December 1993.
9. T. F. Chan and T. Mathew, *Domain decomposition algorithms*, Acta Numerica (1994), 61–143.
10. D. F. D'Azevedo, P. A. Forsyth, and W.-P. Tang, *Toward a cost effective ilu preconditioner with high level fill*, BIT **32** (1992), 442–463.
11. M. Dryja, B. F. Smith, and O. B. Widlund, *Schwarz analysis of iterative substructuring algorithms for elliptic problems in three dimensions*, SIAM J. Numer. Anal. **31** (1994), 1662–1694.
12. M. Dryja and O.B. Widlund, *Some domain decomposition algorithms for elliptic problems*, Iterative Methods for Large Linear Systems (L. Hayes and D. Kincaid, eds.), 1989, pp. 273–291.
13. M. Dryja and O.B. Widlund, *Additive Schwarz methods for elliptic finite element problems in three dimensions*, Fifth Conference on Domain Decomposition Methods for Partial Differential Equations (T. F. Chan, D.E. Keyes, G.A. Meurant, J.S. Scroggs, and R.G. Voit, eds.), 1992.
14. T. Dupont, R. Kendall, and H. Rachford, *An approximate factorization procedure for solving self-adjoint elliptic difference equations*, SIAM J. Numer. Anal. **5** (1968), 558–573.
15. D. J. Evans, *The use of pre-conditioning in iterative methods for solving linear equations with symmetric positive definite matrices*, J. Inst. Maths. Applics. **4** (1968), 295–314.
16. G. Golub and D. Mayers, *The use of preconditioning over irregular regions*, Comput. Meth. Appl. Mech. Eng. **6** (1984), 223–234.

17. T. J. R. Hughes, L. P. Franca, and M. Mallet, *A new finite element formulation for CFD: I. symmetric forms of the compressible Euler and Navier-Stokes equations and the second law of thermodynamics*, Comput. Meth. Appl. Mech. Eng. **54** (1986), 223–234.
18. T. J. R. Hughes and M. Mallet, *A new finite element formulation for CFD: III. the generalized streamline operator for multidimensional advective-diffusive systems*, Comput. Meth. Appl. Mech. Eng. **58** (1986), 305–328.
19. G. Karypis and V. Kumar, *Multilevel k-way partitioning scheme for irregular graphs*, Tech. Report Report 95-064, U. of Minn. Computer Science Department, 1995.
20. D. J. Mavriplis, *A three dimensional multigrid Reynolds-averaged Navier-Stokes solver for unstructured meshes*, Tech. Report ICASE Report 94-29, NASA Langley, 1994.
21. J. A. Meijerink and H. A. van der Vorst, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix*, Math. Comp. **34** (1977), 148–162.
22. J. S. Przemieniecki, *Matrix structural analysis of substructures*, Am. Inst. Aero. Astro. J. **1** (1963), 138–147.
23. Y. Saad, *A flexible inner-outer preconditioned GMRES algorithm*, SIAM J. Sci. Stat. Comp. **14** (1993), no. 2, 461–469.
24. H. A. Schwarz, *Uber einige abbildungensaufgaben*, J. Reine Angew. Math. **70** (1869), 105–120.
25. V. Simoncini and E. Gallopoulos, *An iterative method for nonsymmetric systems with multiple right-hand sides*, SIAM J. Sci. Comput. **16** (1995), no. 4, 917–933.
26. B. Smith, P. Bjorstad, and W. Gropp, *Domain decomposition: parallel multilevel methods for elliptic partial differential equations*, Cambridge University Press, 1996.
27. H. Stone, *Iterative solution of implicit approximations of multidimensional partial differential equations*, SIAM J. Numer. Anal. **5** (1968), 530–558.
28. Y. Wu, X.-C. Cai, and D. E. Keyes, *Additive Schwarz methods for hyperbolic equations*, AMS Cont. Math. **218** (1998).
29. J. Xu, *An introduction to multilevel methods*, Lecture notes: VIIth EPSRC numerical analysis summer school, 1997.

Parallel Turbulence Simulation: Resolving the Inertial Subrange of Kolmogorov's Spectra

Martin Strietzel*, Thomas Gerz

Center for Simulation Software and
Institut für Physik der Atmosphäre
of the German Aerospace Center (DLR)
D-51170 Köln
{martin.strietzel,thomas.gerz}@dlr.de

Abstract. We describe our parallel implementation for large-eddy simulation and direct numerical simulation of turbulent fluids (called PARDISTUF) based on the three-dimensional incompressible Navier - Stokes equation. Benchmark results on a set of european supercomputers under the message-passing platform MPI are presented. Using this programm on a 48 node SP-2 we resolved the inertial subrange of Kolmogorv's turbulence spectra for the first time for a stratified and sheared environmental flow.

1 Introduction

The Institute of Atmospheric Physics at the German Aerospace Research Facility (DLR) in Oberpfaffenhofen is investigating the physics of turbulent fluids. The studies are motivated by the need to understand and predict the diffusion of species concentrations in atmospheric flows which often are turbulent, stably stratified and sheared. One special point of interest is the concern that exhaust gases from aircraft may influence the global climate. The aim of the parallelization activities is to tackle the particular problem of the diffusion properties at small scales.

During the last ten years an extensive program for DIrect numerical (or large-eddy) Simulation of TUrbulent Fluid (DISTUF) at high Reynolds numbers under the influence of shear and stable stratification was developed and optimized for daily use on vector computers such as the Cray Y-MP (cf. [3]). For a simulation run on 128^3 gridpoints with 3000 time steps DISTUF requires about 44 MWords (64bit) of memory and eight hours of CPU time on one processor.

To resolve the inertial subrange of the turbulent energy spectrum the resolution has to be increased to 512^3 gridpoints. This is not feasible on nowadays vector computers. At this point we decided to make our code suitable for state-of-the-art massively parallel systems to access more computing power and more memory. The parallelization is based on the concepts of message-passing and domain decomposition. We use MPI to achieve a high portability.

* Supported by Zentrum für Paralleles Rechnen (University of Cologne)

The successfull parallelization is demonstrated by a simulation run on a 48 processor SP-2 at the DLR. In this run the inertial subrange of Kolmogorov's spectra is resolved for the first time by a computer simulation.

2 Method of simulation

We integrate the time-dependent, incompressible, three-dimensional Navier - Stokes and temperature concentration equations in a rectangular domain and in time. The methods of large-eddy simulation (LES) or direct numerical simulation (DNS) are selectable.

We consider a rectangular domain with coordinates x, y, z or x_i ($i \in \{1,2,3\}$),

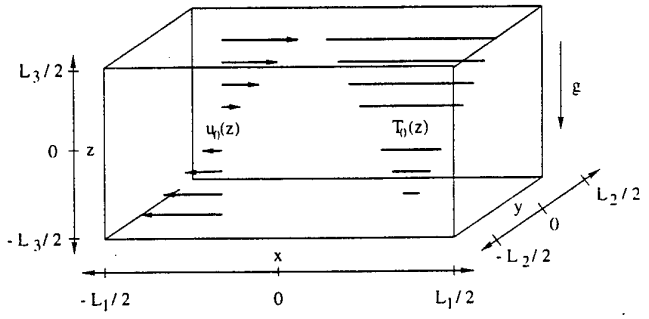


Fig. 1. (a) Simulation domain and (b) mean profiles of velocity and temperature

and side-length L_i. The mean horizontal velocity $U_0(z)$ and mean (reference) temperature $T_0(z)$ possess uniform and constant gradients relative to the vertical coordinate z and are constant in the other directions. The fluid is assumed to have constant molecular diffusivities for momentum and heat. All fields are expressed nondimensionally using $L := L_3$, $\Delta U = \|dU_0/dx_3\|L$ and $\Delta T = \|dT_0/dx_3\|L$ as reference scales for length, velocity and temperature. The turbulent fluctuations relative to these mean values are u_i ($i \in \{1,2,3\}$) for velocity, T for temperature and p for pressure.

The normalized Navier-Stokes-equation, the heat balance, and the continuity equation read

$$\frac{\partial u_i}{\partial t} + \frac{\partial}{\partial x_j}(u_j u_i) + Sx_3 \frac{\partial u_i}{\partial x_1} + Su_3\delta_{i1} = -\frac{\partial \tau_{ij}}{\partial x_j} - \frac{\partial p}{\partial x_i} + Ri\frac{S^2}{s}T\delta_{i3} \quad (1)$$

$$\frac{\partial T}{\partial t} + \frac{\partial}{\partial x_j}(u_j T) + Sx_3 \frac{\partial T}{\partial x_1} + su_3 = -\frac{\partial \tau_{Tj}}{\partial x_j} \quad (2)$$

$$\frac{\partial u_j}{\partial x_j} = 0, \quad (3)$$

where $S = (L/\Delta U)(dU_0/dx_3) \in \{0,1\}$ is the nondimensional shear parameter, $s = (L/\Delta T)(dT_0/dx_3) \in \{-1,0,1\}$ is the stratification parameter, and δ_{ij} is the Kronecker-Delta. Ri is the gradient Richardson number. τ_{ij} and τ_{Tj} denote the diffusive fluxes of momentum and heat. Additionaly the distribution of three passive scalars can be calcultated by solving transport equations for each one. The equations are discretized in an equidistant Eulerian framework using a second-order finite-difference technique on a staggered grid for all the terms in the equations except the mean advection, where pseudo-spectral (Fourier) approximation in x-direction is used. The Adams-Bashforth scheme is employed for time integration of the acceleration terms. The pressure p^{n+1} at the new time-level $n+1$ is obtained by solving the Poisson equation in finite difference form (4), with δ_i as the common finite difference operator and $\tilde{u_i}$ denoting the velocity terms resulting from the Adams-Bashforth scheme.

$$\delta_i \delta_i \, p^{n+1} = \frac{1}{\Delta t} \delta_i \tilde{u_i}. \tag{4}$$

The solution of (4) is obtained using a fast Poisson solver, which includes the shear-periodic boundary condition at time t^{n+1} and applies a combination of fast Fourier transforms and Gaussian elimination. Finally the velocities are updated by the new pressure terms.

3 Parallel approach

Our main aim was to develop a parallel code which is not only efficient, scalable, and numerically correct, but also portable on a wide range of supercomputers. For this reason we decided to take advantage of the MPI message passing standard. The first experiences with the MPI implementations show that this was the right decision. We were able to port the programm to a wide set of computers very fast and without any changes concerning the message passing calls.

3.1 Domain decomposition

Analysing the sequential code we found out that under the aspect of parallelism the one-, two- and three-dimensional fast Fourier transforms (FFT) are the most critical parts of the program. The one-dimensional FFT in x-direction is used in every time step in order to consider the shear flow in the horizontal velocity components. The two-dimensional FFT in x- and y-direction is a part of the Poisson solver for equation (4). For statistic evaluations in the Fourier space, which can be done at user defined intervals, we need a parallel three-dimensional Fourier transform, too.
The best way of implementing two- and three-dimensional FFT on parallel computers is to treat them as a sequence of one-dimensional transforms, which are computed independently on the processors. (cf. [1],[8],[2]). This is no problem at

all on shared memory computers and can be implemented on distributed memory machines by using efficent algorithms for data transposition. For this reason we partition the three-dimensional grid into subdomains allowing to perform a one-dimensional Fourier transform in x-direction without communication. That means that the domain is decomposed in horizontal bars in x-direction (two-dimensional decomposition) or in horizontal (or vertical) planes in x/y (or x/z) direction, what we call one-dimensional decomposition. According to this, the process topology is a one-dimensional or a two-dimensional grid (fig. 2). With the described domain partitioning we can perform all necessary Fourier transforms in y and z direction after data transposition, which can be implemented very efficiently using MPI calls (cf. [6]).

In the Poisson solver we have to solve tridiagonal systems of equations dis-

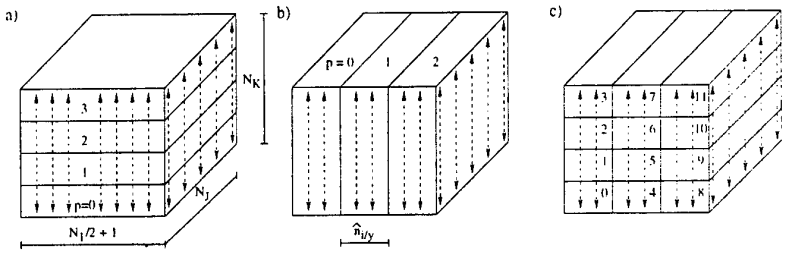


Fig. 2. Datadecomposition: One-dimensional (a, b) and two-dimensional (c). The arrows show the distribution of the tridiagonal systems (5)

tributed in z-direction. This is another point where we have to think about an efficient parallel algorithm. Our approach for this is described in chapter 3.2. All other parts of the algorithm can be calculated independently on all processors, if we ensure an overlap of one row or column of gridpoints in each direction, which has to be updated after each time step.

3.2 Parallel Poisson solver

For pressure correction we have to solve the Poisson equation

$$\frac{\partial \tilde{u}_i(x)}{\partial x_i} = \Delta t \frac{\partial^2 p}{\partial x_i^2}$$

for the pressure p in each time step. This is done by two-dimensional Fourier transforms of the left handsides in both horizontal directions, which results in $(N_I/2+1) * N_J$ ($N_{I,J,K}$ = number of gridpoints in x, y, and z direction) tridiagonal systems of equations of rank N_K with shear-periodic boundaries (ω is a

System (manufacturer)	Processor (manufacturer)	Max. number of proc.	Memory[2] p. proc. (MB)	Performance per proc. peak (MFlop/s)	SPEC fp92[3]	cache (KB)
Massivly-parallel systems with distributed memory:						
GC/PowerPlus (Parsytec)	601+ (PowerPC)	192 (96)	32 (64)	80	125	32
SP2 (IBM)	Power2 (IBM)	58 (+8)	256 (128)	267 (133)	244.6 (202.1)	256 (128)
T3D (Cray)	alpha 21064 (DEC)	512	64	150	200	8
Systems with shared memory:						
Power Challenge XL (SGI/SNI)	R8000 (MIPS)	16	500	300	311	4.000
Ultra Enterprise 4000 (SUN)	Ultra I (SPARC)	8	128	unknown to the author	386	512
Parallel vector machines:						
J916 (Cray)		16	256	200		

Table 1. Technical data of the parallel systems

complex shear factor):

$$\hat{p}_{i-1} - 2\hat{p}_i + \hat{p}_{i+1} = \hat{u}_i\,, \qquad i = 1, 2, \ldots, N_K \quad \hat{p}_i, \hat{u}_i \in C \tag{5}$$
$$\hat{p}_0 = \omega \hat{p}_{N_K}\,, \qquad \hat{p}_{N_K+1} = \omega^{-1}\hat{p}_1$$

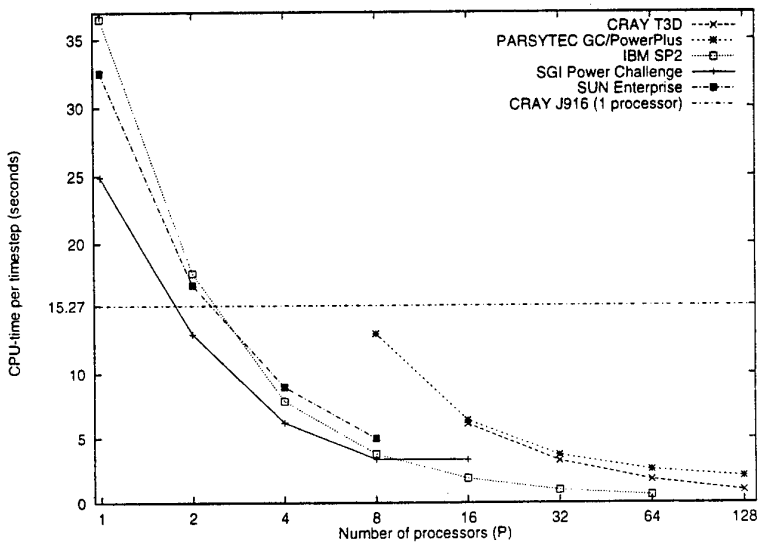
The components $\hat{p}_i$, $\hat{u}_i$ of each of these systems are distributed in z-direction on the grid. The systems themselves are distributed in x- and y-direction (fig. 2). After solving the equations the results are transformed back, and we finally get the pressure terms p for the next time step.
For parallelization we must distinguish between the phase of Fourier transforms and the algorithm for solving the tridiagonal systems. The Fourier transforms can be done simultaneously on the distributed datasets. In case of the two-dimensional decomposition we have to include a data transposition step.
The distributed equations can be solved independently by the subsets of processes with the same x and y coordinates. Each of the in z-direction distributed systems (5) is solved by an improved divide & conquer method (cf. [7]) based on an algorithm from *Mehrmann* (cf.[5]).

4 Experiences on parallel systems

We had the opportunity to run the parallel turbulence simulation code on a set of parallel computers, including shared memory and distributed memory sys-

tems. The machines are summarized in table 1.
The parallel test runs are compared with a run on one processor of the CRAY J916 with 256 MB of memory and a peak performance of 200 MFlops. This is the machine the original code is designed and highly optimized for.
In order to evaluate performance data on all machines we run simulations with

Fig. 3. Runtimes for one timestep on 128^3 gridpoints.



shear, stratification, and three passive scalars about 64 time steps on 128^3 gridpoints. Here we discuss the timings for one timestep in this run.
On the shared memory systems we get a speed-up of 7.44 on eight SGI processors and one of 6.56 on eight SUN CPUs. On more than 8 processors of the Power Challenge this value couldn't be increased. This restriction depends on the fixed bandwidth of the underlying communication system, the SGI data bus.
A totaly different picture is the speed-up on the distributed systems. The timing results are presented in figure 3. Only on the SP2 the 128^3 grid fits on one processor and here we get a speed-up of 62.95 for 64 processors.
The best timing results for one timestep has the SP2, which with 16 processors is 3.5 times faster than the Parsytec GC or the Cray T3D. The T3D with 128 CPUs is still 2 times slower than 64 SP2 processors, but 2 times faster than the Parsytec GC.
On the IBM SP2 and the SUN Enterprise we need at least three processors to get the same performance than one CRAY J90 vector processor. On the SGI Power Challenge two CPUs already are faster than the J90.

The results show that our parallel code has a good efficiency on medium sized shared memory and larger distributed memory machines. But not only the performance is important. Running the programm on 48 fat nodes of the SP2 allows as to use 12 Gbyte memory. Therfore we are able to compute grids with 480^3 gridpoints, this is more than 8 times larger than on vector machines.

5 Resolving the Inertial Subrange

In the Kolmogorov spectra of turbulence energy [4], the kinetic energy in the flow is plotted over all wavenumbers, which are reciprocal to the size of the eddy-structurs. This spectra can be divided into three subranges: production, inertia, and dissipation. In the inertial subrange the flow has universial properties, which do not depend on the geometrie or other physical parameters of the flow. The existence of the inertial subrange depends on the Reynolds number of the flow. In direct numerical simulation this is directly correlated to the grid resolution. On the SP2 we were able to manage one run on 480^3 gridpoints with a Reynolds

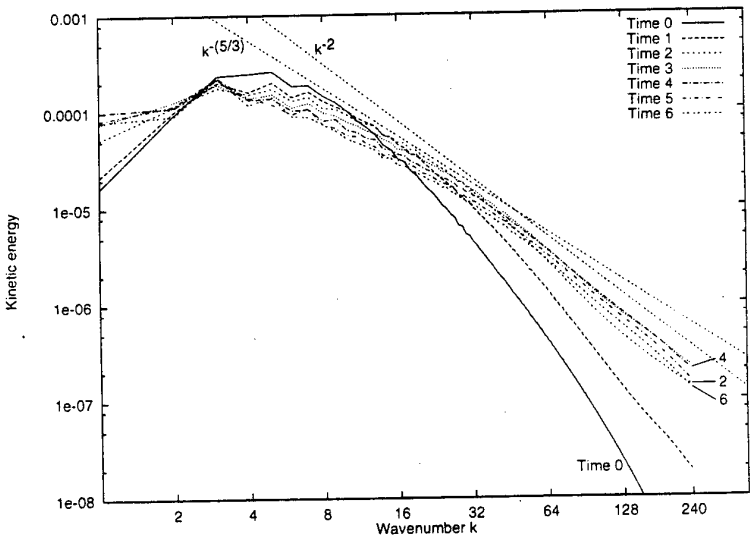


Fig. 4. Spectra of the kinetic energy of the simulation with 480^3 gridpoints, Reynolds number 600 (based on velocity fluctuation and integral scale)

number of 600. By using a gradient Richardson number of 0.13 we force the flow to become stationary, that means the kinetic energy becomes constant. After an initial phase this flow shows a self similar energy spectra with a decay as $k^{-5/3}$

for $20 < k < 50$. This is the main indication for a resolved inertial subrange. For our knowledge, this is the first time that the Kolmogorv's inertial subrange could be resolved in a computer simulation.

6 Conclusion

A fully parallelized version of an incompressible turbulence simulation code has been presented. The parallel code achieves 2.446 GFlop/s on 64 SP2 processors, this is 26.3 times faster than on one J90 vector processor. Due to the message passing standard MPI we achieved a perfect portability. The developed code is now fitted for the use on state-of-the-art parallel computers.
We have demonstrated that the concept on message passing is not exclusive for distributed memory machines and we have shown that a MPI implementation on a few processors, which derives advantage from the fast communication possibilities on shared memory machines, can be better suited for parallel computing than on some other dedicated message passing machines.
On the other hand we developed a parallel code which already leads to new physical results. First in the world, we resolved the inertial subrange of a homogeneously turbulent and stratified shear flow by a direct numerical simulation. This demonstrates the way how parallel computing can open the door for new fundamental results in physics.

References

1. Martin Bücker. Zweidimensionale Schnelle Fourier–Transformation auf massiv parallelen Rechnern. Technical Report Jül-2833, ZAM, Forschungszentrum Jülich, D-52425 Jülich, November 1993.
2. Clare Yung-Lei Chu. *The fast Fourier transform on hypercube parallel computers.* PhD thesis, Cornell University, 1988.
3. Thomas Gerz, Ulrich Schumann, and S. E. Elghobashi. Direct numerical simulation of stratified homogeneous turbulent shear flows. *J. Fluid Mech.*, 200:563–594, 1989.
4. A. N. Kolmogorov. The local structure of turbulence in incompressible viscous fluid for very large Reynolds number. *C. R. Acad. Nauk SSSR*, 30:301–303, 1941. Reprinted in Proc. Soc. Lond. A, 434, 9-13(1991).
5. Volker Mehrmann. Divide and conquer methods for block tridiagonal systems. *Parallel Comput.*, 19:257–279, 1992.
6. Message Passing Interface Forum. *MPI: A message-passing interface standard.* June 1995.
7. Ulrich Schumann and Martin Strietzel. Parallel solution of tridiagonal systems for the Poisson equation. *J. Sci. Comput.*, 10(2):181–190, Juni 1995.
8. Paul N. Swarztrauber. Multiprocessor FFTs. *Parallel Comput.*, 5:197–210, 1987.

This article was processed using the LaTeX macro package with LLNCS style

A Systolic Algorithm for the Factorisation of Matrices Arising in the Field of Hydrodynamics

S.-G. Seo[1], M. J. Downie[1], G. E. Hearn[1] and C. Phillips[2]

[1]Department of Marine Technology, University of Newcastle upon Tyne, Newcastle upon Tyne, NE1 7RU, UK

[2]Department of Computing Science, University of Newcastle upon Tyne, Newcastle upon Tyne, NE1 7RU, UK

Abstract. Systolic algorithms often present an attractive parallel programming paradigm. However, the unavailability of specialised hardware for efficient implementation means that such algorithms are often dismissed as being of theoretical interest only. In this paper we report on experience with implementing a systolic algorithm for matrix factorisation and present a modified version that is expected to lead to acceptable performance on a distributed memory multicomputer. The origin of the problem that generates the full complex matrix in the first place is in the field of hydrodynamics.

1. Forming the Linear System of Equations

The efficient, safe and economic design of large floating offshore structures and vessels requires a knowledge of how they respond in an often hostile wave environment [1]. Prediction of the hydrodynamic forces experienced by them and their resulting responses, which occur with six rigid degrees of freedom, involves the use of complex mathematical models leading to the implementation of computationally demanding software. The solution to such problems can be formulated in terms of a velocity potential involving an integral expression that can be thought of as representing a distribution of sources over the wetted surface of the body. In most applications there is no closed solution to the problem and it has to be solved numerically using a discretisation procedure in which the surface of the body is represented by a number of panels, or facets. The accuracy of the solution depends on a number of factors, one of which is the resolution of the discretisation. The solution converges as resolution becomes finer and complicated geometries can require very large numbers of facets to attain an acceptable solution.

In the simplest approach a source is associated with each panel and the interaction of the sources is modelled by a Green function which automatically satisfies relevant 'wave boundary conditions'. The strength of the sources is determined by satisfying a velocity continuity condition over the mean wetted surface of the body. This is

achieved by setting up an influence matrix, A, for the sources based on the Green functions and solving a linear set of algebraic equations in which the unknowns, **x**, are either the source strengths or the velocity potential values and the right-hand side, **b**, are related to the appropriate normal velocities at a representative point on each facet. For a given wave frequency, each facet has a separate source contributing to the wave potential representing the incoming and diffracted waves, ϕ_0 and ϕ_7, and one radiation velocity potential for each degree of freedom of the motion, ϕ_i : i=1,2,...,6. When the velocity potentials have been determined, once the source strengths are known, the pressure can be computed at every facet and the resultant forces and moments on the body computed by integrating them over the wetted surface. The forces can then be introduced into the equations of motion and the response of the vessel at the given wave frequency calculated.

The complexity of the mathematical form of the Green functions and the requirement to refine the discretisation of the wetted surfaces within practical offshore analyses, significantly increases the memory and computational load associated with the formulation of the required fluid-structure interactions. Similarly the solution of the very large dense square matrix equations formulated in terms of complex variables requires considerable effort to provide the complex variable solution. In some problems 5,000 panels might be required using constant plane elements or 5,000 nodes using higher order boundary elements, leading to a matrix with 25,000,000 locations. Since double precision arithmetic is required, and the numbers are complex, this will require memory of the order of 4 gigabytes. The number of operations for direct inversion or solution by iteration is large and of the order of n^3, e.g. 5,000 elements requires 125,000 x 10^6 operations. Furthermore, the sea-state for a particular wave environment has a continuous frequency spectrum which can be modelled as the sum of a number of regular waves of different frequencies with random phase angles and amplitudes determined by the nature of the spectrum. Determination of vessel responses in a realistic sea-state requires the solution of the boundary integral problem described above over a range of discrete frequencies sufficiently large to encompass the region in which the wave energy of the irregular seas is concentrated. In view of the size and complexity of such problems, and the importance of being able to treat them, it is essential to develop methods to speed up their formulation and solution times. One possible means of achieving this is through the use of parallel computers [2] [3].

Since A is a full, square matrix, and in view of the uncertainty regarding the convergence of iterative methods, the use of a direct method of solution based on elimination techniques would seem the most attractive proposition. The method of LU-decomposition has been chosen because in this scheme only one factorisation is required for multiple unknown right-hand side vectors **b**. It is well known that this factorisation is computationally intensive, involving order n^3 arithmetic operations (multiplications and additions). In contrast the forward- and backward-substitution required to solve the original system once the factorisation has been performed involves an order of magnitude less computation, namely order n^2, which becomes insignificant as the matrix size increases. Consequently, we limit consideration to the factorisation process only.

Formally, we have that the elements u_{ij} of U are given by

$$u_{rj} = a_{rj} - \sum_{k=1}^{r-1} l_{rk} u_{kj} \qquad j = r,\ldots\ldots,n \tag{1}$$

and l_{ij} of L are given by

$$l_{ir} = \left(a_{ir} - \sum_{k=1}^{r-1} l_{ik} u_{kr} \right) / u_{rr} \qquad i = r+1,\ldots,n \tag{2}$$

(Doolittle factorisation) leading to an upper-triangular U and a unit lower-triangular L.

2. A Naive Systolic Algorithm Solution

A systolic system can be envisaged as an array of synchronised processing elements (PEs), or cells, which process data in parallel by passing them from cell to cell in a regular rhythmic pattern. Systolic arrays have gained popularity because of their ability to exploit massive parallelism and pipelining to produce high performance computing [4] [5]. Although systolic algorithms support a high degree of concurrency, they are often regarded as being appropriate only for those machines specially built for the particular algorithm in mind. This is because of the inherent high communication/computation ratio.

In a soft-systolic algorithm, the emphasis is on retaining systolic computation as a design principle and mapping the algorithm onto an available (non-systolic) parallel architecture, with inevitable trade-offs in speed and efficiency due to communication and processor overheads incurred in simulating the systolic array structure.

Initially a systolic matrix factorisation routine was written in Encore Parallel Fortran (epf) and tested on an Encore Multimax 520. This machine has seven dual processor cards (a maximum of ten can be accommodated), each of which contains two independent 10 MHz National Semiconductor 32532 32-bit PEs with LSI memory management units and floating-point co-processors. This work was undertaken on a well-established machine platform that was able to provide a mechanism for validating the model of computation, and the software developed from that model, with a view to refining the model for subsequent development on a state-of-the-art distributed memory parallel machine. epf's parallel program paradigm is much simpler to work with than that for message passing on distributed memory architectures and produces a convenient means of providing validation data for subsequent developments. As long as the underlying algorithm is retained this implementation can be used to check the correctness of the equations developed and the validity of the algorithm with a view to an eventual port.

The task of systolic array design may be defined more precisely by investigating the cell types required, the way they are to be connected, and how data moves through the array in order to achieve the desired computational effect. The hexagonal shaped cells employed here are due to Kung and Leiserson [6].

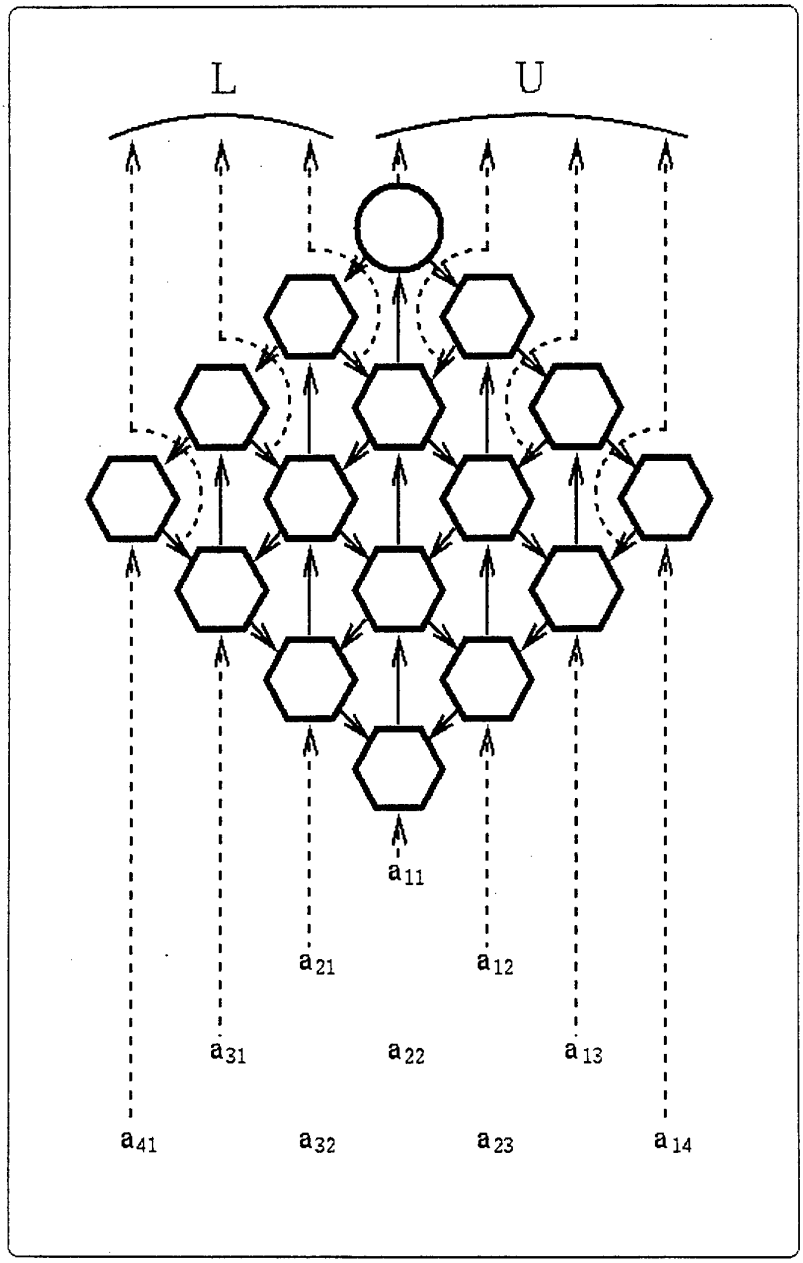


Fig. 1. Data flow for LU factorisation

The manner in which the elements of L and U are computed in the array and passed on to the cells that require them is demonstrated in Fig. 1. An example of a 4x4 cell array is shown for the purposes of illustrating the paradigm employed. Of the PEs on the upper-right boundary, the top-most has three roles to perform:

1. Produce the diagonal components of L (which, for a Doolittle-based factorisation, are all unit).
2. Produce the diagonal components of U using elements of A which have filtered through the cells along the diagonal (and been modified, as appropriate).
3. Pass the reciprocal of the diagonal components of U down and left.

The cells on the upper left boundary are responsible for computing the multipliers (the elements of L), having received the appropriate reciprocals of the diagonal elements of U.

The flow of data through the systolic array is shown in Fig. 1. Elements of A flow in an upward direction; elements of L (computed from (**1**)) flow in a right-and-down direction; and elements of U (computed from (**2**)) flow in a left-and-down direction. Each cell computes a value every 3 clock ticks, although they start at different times. Note that at each time step each cell responsible for forming an element of L or U calculates one multiplication only in forming a partial summation. Data flowing out correspond to the required elements of the L and U factors of A. Formally, the elements of A are fed into the cells as indicated, although for efficiency the cells are directly assigned the appropriate elements of A.

Table 1 shows the execution times (T_p) of the parallel code with a fixed-size (100*100 double complex) matrix and various numbers of PEs (p). The gradual algorithmic speed-up (S_p), defined as the ratio of the time to execute the program on p processors to the time to execute the same parallel program on a single processor, is clearly seen all the way up to twelve PEs. The (generally) decreasing efficiency (E_p), defined as the ratio of speed-up to the number of PEs times 100, is a consequence of the von Neumann bottleneck. The results show some minor anomalies, but this is not atypical when attempting to obtain accurate timings on a shared resource, with other processes - system or those of other users - interfering with program behaviour, even at times of low activity. At this level, the results are encouraging.

Table 1. Shared memory implementation

p	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
T_p	48.9	28.3	20.2	14.9	12.3	10.6	8.9	8.0	7.3	6.6	6.2	6.1
S_p	1	1.7	2.4	3.3	4.0	4.6	5.5	6.1	6.7	7.4	7.9	8.0
E_p	100	87	81	82	80	77	79	77	74	74	72	68

The algorithm was compared with an existing parallel matrix factorisation routine [7], which uses more conventional techniques, available in the Department of Marine Technology (DMT). The results are summarised in Fig 2, where $S_{p(\text{systolic})}$ denotes the speedup for the systolic algorithm (from Table 1) and $S_{p(DMT)}$ the speedup

for the DMT algorithm. The extra cost of the systolic algorithm due to overheads

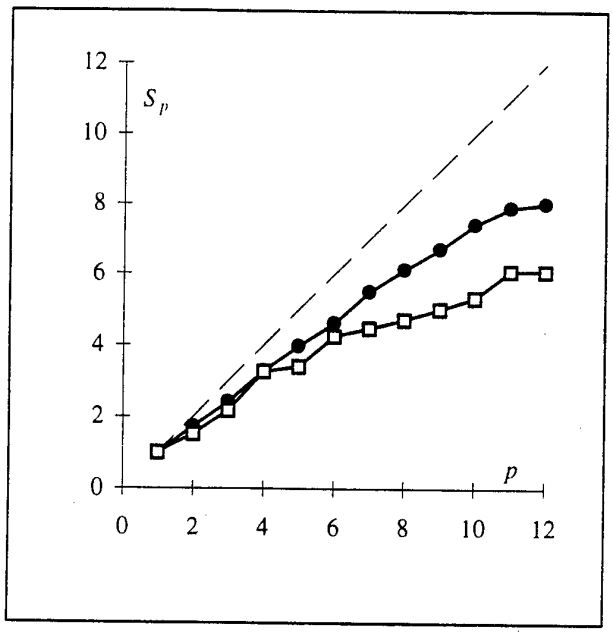


Fig. 2. Comparison of speedup for systolic and DMT algorithms.

associated with index calculation and array accesses causes significant delays in the computation resulting in much poorer performance of the systolic algorithm in comparison to the DMT routine in terms of elapsed time. Nevertheless, the systolic algorithm shows better speedup characteristics than the DMT algorithm, as illustrated by Fig. 2.

If A is an n by n dense matrix then the systolic algorithm implemented on n^2 PEs can compute L and U in $4n$ clock ticks, giving a cell efficiency of 33%. Assuming a hardware system in which the number of PEs is much less than the number of cells, and using an appropriate mapping of cells to PEs, we can improve this position considerably, and we now address this issue.

3. An Improved Systolic Algorithm

As already indicated, the ultimate goal is to produce an efficient systolic matrix factorisation routine for general-purpose distributed parallel systems including clusters of workstations. This is to be achieved by increasing the granularity of the computation within the algorithm and thus reducing the communication/ computation ratio, while balancing the load on each PE to minimise the adverse

effect due to enforced synchronisation. Nevertheless, the characteristics peculiar to systolic algorithms should be retained. Thus we aim at

- massive, distributed parallelism
- local communication only
- a synchronous mode of operation

Each PE will need to perform far more complex operations than in the original systolic systems used as the original basis for implementation. There is an inevitable increase in complexity from the organisation of the data handling required at each synchronisation (message-passing) point.

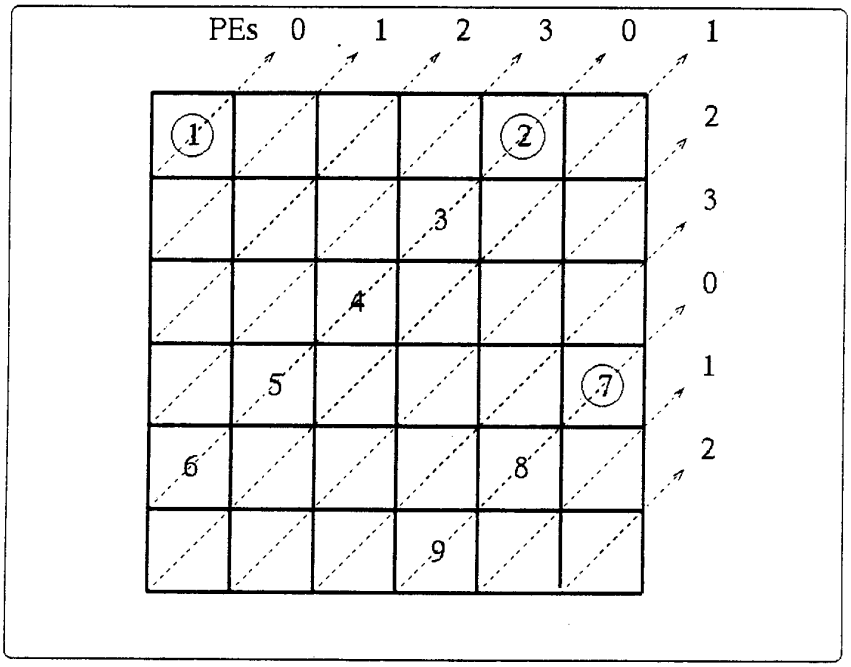


Fig. 3. Allocation of pseudo cells to PEs for a 6x6 matrix.

The systolic algorithm can be regarded as a wave front passing through the cells in an upwards direction in Fig. 1. This means that all pseudo cells in a horizontal line, corresponding to a reverse diagonal of A, become active at once. It follows that we allocate the whole of a reverse diagonal to a PE, and distribute the reverse diagonals from the top left to the bottom right in a cyclic manner so as to maintain an even load balance (for a suitably large matrix size). Fig. 2 shows such a distribution for a 6x6 matrix distributed on 4 PEs.

The computation starts at PE0 with pseudocell 1. As time increases, so the computation domain over the matrix domain increases, and later shrinks. The shape of the computation domain is initially triangular, to include the first few reverse diagonals. On passing the main reverse diagonal, the computation domain becomes pentagonal, and remains so until the bottom right-hand corner is reached, when it becomes a quadrilateral. Once the computation domain has covered the whole of the domain of pseudo cells it shrinks back to the top left, whilst retaining its quadrilateral shape. The whole process is completed in $3n-2$ timesteps.

A Fortran implementation of the revised systolic algorithm is currently under development using the distributed memory Cray T3D at the Edinburgh Parallel Computer Centre (EPCC), and the Fujitsu AP1000 at Imperial College, London, and MPI [8] [9] for message passing.

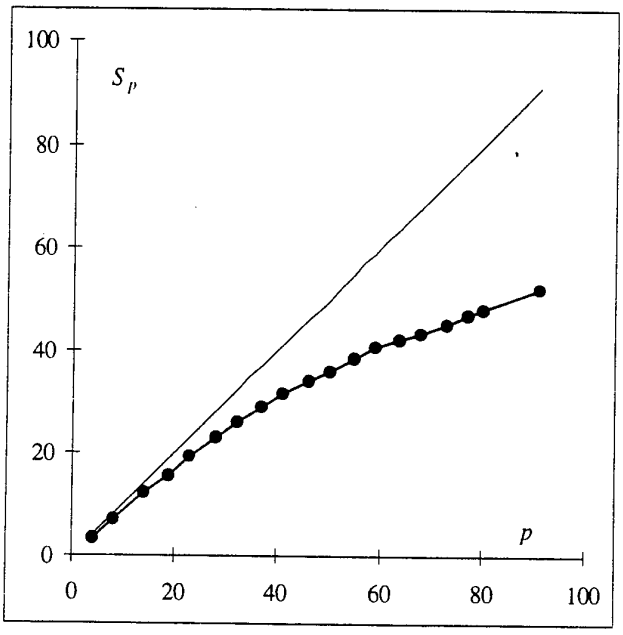


Fig. 4. Speedup on distributed memory machine for 400 by 400 array

In this implementation the elements in each reverse (top-right to bottom-left) diagonal of the matrix are bundled together so that they are dealt with by a single PE and all of the reverse diagonals which are active at a given time step are again grouped together to be passed around as a single message. Preliminary experience with the revised algorithm indicates that it scales well, as shown by the speed up figures for a 400 by 400 array computed on the Fujitsu machine and illustrated in Fig. 4.

4. Conclusions

The algorithm described initially was a precursor to a more generalized one designed with the intention of increasing the granularity of computation and with the solution of large systems of equations in mind. As expected, it performed poorly in terms of elapsed time due to the penalties imposed by an unfavourable balance between processor communication and computation. However, the speedup characteristics compared favourably with those of a more conventional approach and pointed to the potential for the generalised algorithm.

Accordingly, a generalised version of the algorithm has been developed and is currently being tested. The speedup obtained on the distributed memory machine suggest that the approach taken is valid. It remains to benchmark the algorithm against a conventional solver, such as the one available within the public domain software pachage, ScaLAPACK.

References

1. Hearn, G.E., Yaakob, O., Hills, W.: Seakeeping for design: identification of hydrodynamically optimal hull forms for high speed catamarans. Proc. RINA Int. Symp. High Speed Vessels for Transport and Defence, Paper 4, London (1995), pp15.
2. Hardy, N., Downie, M.J., Bettess, P., Graham, J.M.: The calculation of wave forces on offshore structures using parallel computers., Int. J. Num. Meth. Engng. **36** (1993) 1765-1783
3. Hardy, N., Downie, M.J., Bettess, P.: Calculation of fluid loading on offshore structures using parallel distributed memory MIMD computers. Proceedings, Parallel CFD, Paris (1993).
4. Quinton, P., Craig, I.: Systolic Algorithms & Architectures. Prentice Hall International (1991)
5. Megson, G. M. (ed.): Transformational Approaches to Systolic Design. Chapman and Hall (1994)
6. Kung, H. T. and Leiserson, C. E. Systolic Arrays for VLSI, Sparse Matrix Proceedings, Duff, I.S. and Stewart, G.W. (ed.). Society for Industrial and Applied Mathematics (1979) PP.256-282.
7. Applegarth, I., Barbier, C., Bettess, P.: A parallel equation solver for unsymmetric systems of linear equations. Comput. Sys. Engng. **4** (1993) 99-115
8. MPI: A Message-Passing Interface Standard. Message Passing Interface Forum (May 1994)
9. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI: The Complete Reference. MIT Press (1996)

The study of a parallel algorithm using the laminar backward-facing step flow as a test case

P.M. Areal[1] and J.M.L.M. Palma[2]

(1) Instituto Superior de Engenharia do Porto
Rua de São Tomé
4200 Porto, Portugal
(e-mail: *pareal@sfc6.fe.up.pt*)

(2) Faculdade de Engenharia da Universidade do Porto
Rua dos Bragas
4099 Porto Codex, Portugal
(e-mail: *jpalma@fe.up.pt*)

Abstract. The current study discusses the results of parallelization of a computer program based on the SIMPLE algorithm and applied to the prediction of the laminar backward-facing step flow. The domain of integration has been split into subdomains, as if the flow were made up of physically distinct domains of integration.
The convergence characteristics of the parallel algorithm have been studied as a function of grid size, number of subdomains and flow Reynolds number. The results showed that the difficulties of convergence increase with the complexity of the flow, as the Reynolds number increases and extra recirculation regions appear.

1 Introduction

Parallel computing has become more and more common, and has developed from a subject of specialized scientific meetings and journals into an affordable technology, to a point where we feel that no references are needed to support this statement.

Fluid flow algorithms are complex, given the number of equations involved, the elliptic nature, non-linearity and peculiarities of the pressure-velocity coupling for incompressible flows. All these features are familiar to those in the fluid dynamics community and make this a formidable set of equations intractable by theoretical approaches. The parallelization makes things even worse and our option was to study parallel fluid algorithm through applications to a series of flow geometries and conditions.

In the present study we discuss the numerical behaviour of a parallel version of the SIMPLE algorithm [14]. We show how the convergence was influenced by the Reynolds number, grid size, number of subdomains and overlapping between the subdomains. A previous work [3] has been followed and new findings and conclusions were added as a result of a new set of calculations using different flow geometry and conditions, i.e. the laminar backward-facing step flow.

2 Mathematical model and strategy of parallelization

The fluid flow equations, assuming steady-state, incompressible and Newtonian fluid, were solved on a Cartesian coordinate system.

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{1}$$

$$\rho \frac{\partial u_i u_j}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \frac{\partial}{\partial x_j}\left[\mu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)\right] \tag{2}$$

where u_i is the velocity along the direction x_i, p is the pressure, and ρ and μ are the density and fluid dynamic viscosity, respectively.

These equations were discretized in a numerical grid, with all variables being defined at the same location (the collocated grid [15]) and following the finite volume approach [6]. The hybrid and central finite differencing schemes were used for discretization of the convective and diffusive terms, respectively.

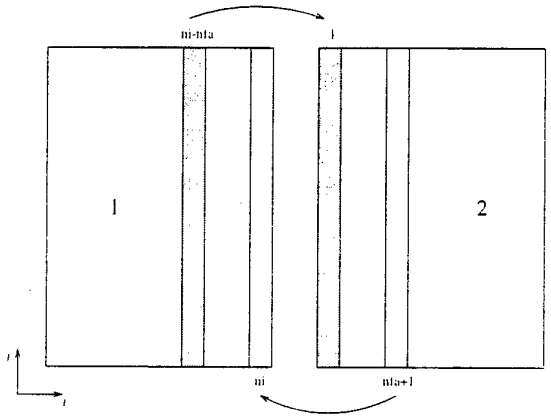


Fig. 1. Overlapping and data exchange between subdomains

The algorithm entitled SIMPLE [14], was used. An approach where, after rewriting the continuity equation (1) as a function of a pressure correction variable, mass and momentum conservation are alternately enforced. Equations (1) and (2) are solved as if they were independent (or segregated) systems of equations, until a prescribed criterion of convergence can be satisfied.

For parallelization of the algorithm, the integration domain was split along the horizontal direction into a variable number of subdomains, with overlapping (Figure 1). Within each subdomain, the SIMPLE algorithm was used, followed by exchange of data (pressure gradient, and both u and v velocities) between subdomains. This was one iteration; equivalent to one iteration of the sequential version of the SIMPLE algorithm, comprising the full domain.

The simplest case, with 2 subdomains only, will be used as an example (Figure 1). The first column of subdomain 2 (the west boundary condition) was taken as one of the columns interior to subdomain 1. The last column of subdomain 1 (the east boundary condition) was taken as one of the columns interior to subdomain 2. The level of overlapping between the subdomains (`nfa`) depended on where inside the subdomains the interior columns were located. The amount of transferred data did not depend on the overlapping: that was a function of the number of grid nodes along the vertical only.

Other strategies of parallelization have been suggested involving parallelization of the equation solver only, either in case of a segregated approach (e.g.: [11]) or in cases where all the fluid equations are solved simultaneously as part of a single system (e.g.: [5]). Either of these two alternatives, when compared with the present parallelization strategy, improve robustness at a cost of increased communication times.

The test case was the laminar backward-facing step flow (Figure 2), with an expansion ratio of 1:2 and a domain size extended over 30 step height, h. A parabolic velocity profile was specified at the inlet section. At the walls, velocities were set to zero and the pressure was found by zero gradient along the perpendicular direction. Zero axial gradient was the boundary condition at the outlet section for all variables.

The tri-diagonal matrix algorithm (TDMA) was used to solve the system of algebraic linearized equations, with under-relaxation factors of 0.7 for velocities and 0.3 for pressure. The number of TDMA iterations was fixed and set at 2 for u and v velocities, and 4 for pressure. There was no previous optimization of these parameters, which were kept unaltered during the course of the current study.

The calculations were stopped when the residual was lower than 10^{-4}.

3 Results

Results were obtained for numerical grids ranging from 100×64 up to 250×160, and Reynolds number between 10^2 and 10^3 in steps of 10^2. The Reynolds number was defined by $Re=\rho U 2h/\mu$, where U is the average axial velocity at the inlet section. The flow regime remains laminar for Reynolds number lower than 1200 (cf. [2].

The calculations were performed on a shared memory computer architecture with a 1.2 G_2B/s system bus bandwidth (SGI Power Challenge, with 4 processors R8000/75 MHz and a total of 512 M_2B of RAM; operating system IRIX 6.1 and fortran compiler MIPSPro Power Fortran77 6.1). The communication between processors was performed via version 3.3 of PVM message passing protocol [7].

3.1 The flow pattern

For a prior assessment of the computer program, the streamline pattern (Figure 2) was analyzed. For all Reynolds number being tested, downstream of the step,

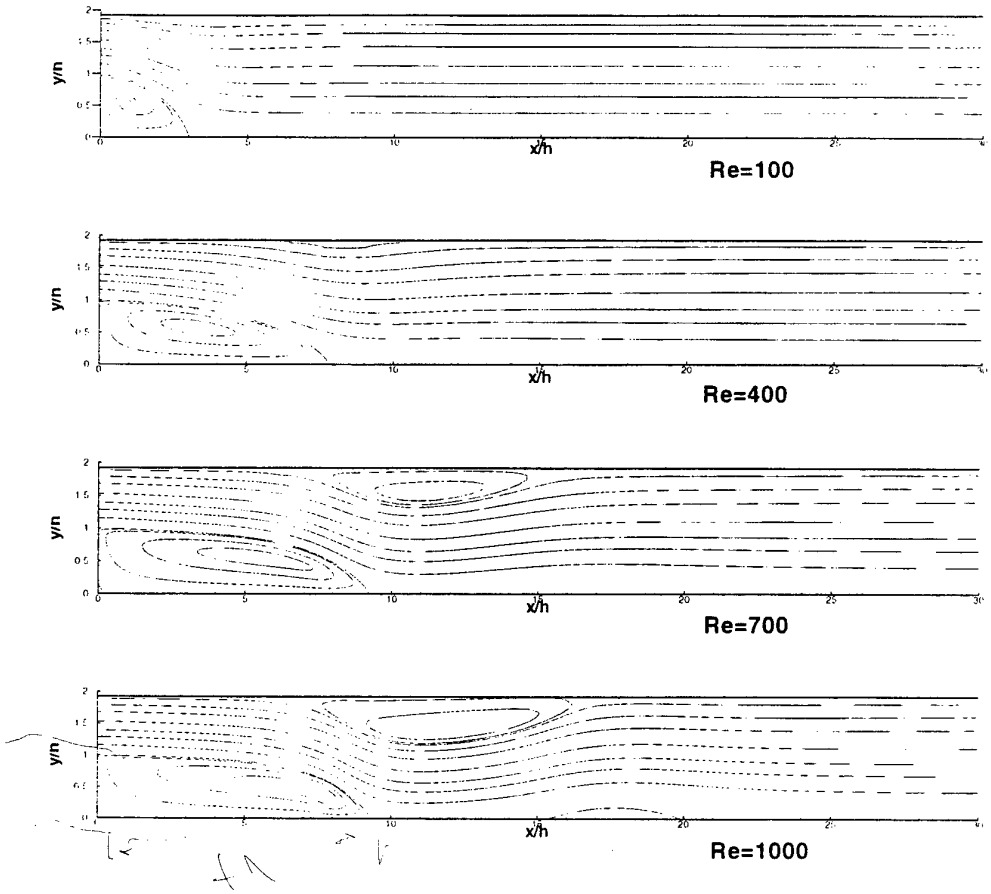


Fig. 2. Streamline pattern as a function of the Reynolds number

there was a main recirculation region, whose length increased with the Reynolds number (Figure 3). The results follow the trend as observed in experiments (cf. [2], [16]) and previous calculations (eg. [10], [18], [4]). The deviation from the experimental curve at Re=500 has been attributed to three-dimensional effects, not accounted for by our calculations.

For a Reynolds number around 400, attached to the top wall of the channel, a second recirculation appears (Figure 2), reaching a maximum length of about $10h$ at Re=1000 (the highest Reynolds number being used). This is a flow pattern that has been observed experimentally (cf. [2], [16]) and is shown here to confirm the quality of our calculations.

Around 50% of the computing time was spent within the routine for solu-

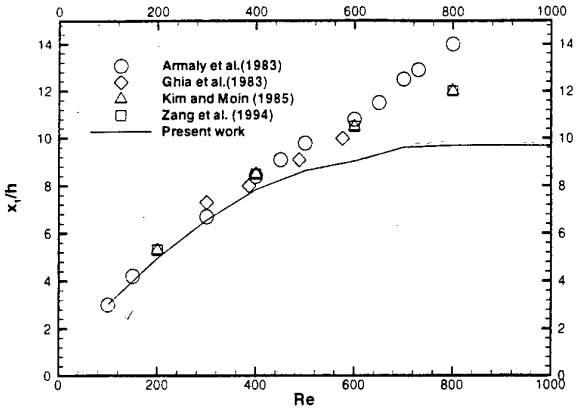


Fig. 3. Size of the main recirculation region as a function of the Reynolds number

tion of the linear system of equations. The assembling of the coefficients of the three differential equations of pressure, and u and v velocities required 48%. The communication time was estimated to be 2% of the total computing time. This was a consequence of the computer architecture, but also a consequence of the algorithm, with the communication overhead much reduced compared with parallel fluid algorithms based on the parallelization of the equation solver.

3.2 The convergence of the parallel algorithm

We were interested in studying the effect of parallelization on the convergence of the algorithm and Figure 4 shows the number of iterations as a function of Reynolds number for four grid sizes. In case of the parallel version the results have been plotted in terms of global iteration and the equivalent grid size of the sequential version. There are two regions in this Figure.

In region I the number of iterations decreased to a minimum, obtained at Re=400 for grids 200×128 and 250×160, and Re=500 for grids 100×64 and 150×96. This is related with the recirculation region attached to the top wall and can be confirmed by joint observation of Figures 4 and 2.

For Reynolds number higher than 400 (or 500 for grids 100×64 and 150×96), in region II, the number of iterations increased with the Reynolds number. The convergence becomes more difficult as the recirculation in the top wall increases (see also Figure 2). The convergence is tightly coupled with the flow pattern.

There was a minimum value of the Reynolds number (region I), depending on the grid size, for which the residual did not fall below 10^{-4} and the convergence criterion was not satisfied. For instance, for Re=100 after 9000, 6000 and 3550 iterations, for grids 250×160, 200×128 and 150×96, the residual became constant at 3.28×10^{-4}, 1.97×10^{-4} and 1.12×10^{-4}, respectively. For coarser grids, the

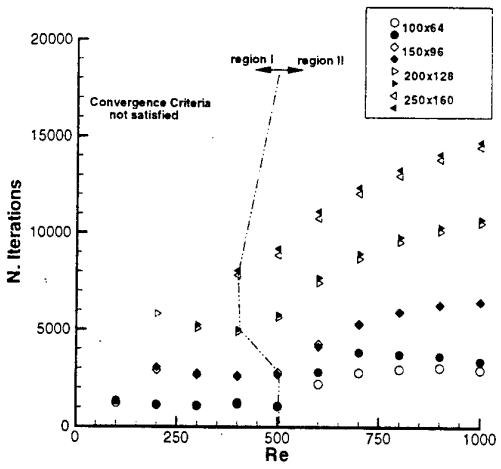


Fig. 4. Number of iterations as a function of the Reynolds number and grid size (filled symbols: parallel version (2 subdomains); open symbols: sequential version)

constant residual was lower and could be reached at a lower number of iterations. However, the flow pattern was as expected and we concluded that the criterion of convergence was too restrictive. Nevertheless, and for consistency with the remaining cases the criterion of convergence was not changed and we considered these as non-converged solutions. To some extent, this is related with the numerical technique used to solve the linear system of equations. For instance, calculations performed using the SIP (*Strong Implicit Procedure* [17]), solver were able to satisfy the convergence criterion for grid 150×96, Re=100.

The results of the parallel version with 2 subdomains (filled symbols) are also included in Figure 4. In most of the cases, the number of iterations (global iterations) of the parallel was identical to the number of iterations of the sequential version. However, there were cases where the parallel version converged faster (e.g. grid 150×96 and Re=600), whereas in the most unfavourable situation (grid 100×64 and Re=700), the parallel required 38% more iterations than the sequential version.

The convergence characteristics of grids 100×64 and 200×128 are shown in Figure 5 as a function of the Reynolds number. In case of grid 100×64 (Figure 5a) for Re lower than 500 the sequential requires more iterations to converge than the parallel version, whereas for Re higher than 500 the situation is reversed. In general, and for Re higher than 500 the number of iterations increases with the number of subdomains. In case of Re=1000 the calculation with 3 subdomains requires 3100 iterations compared with 2800 of the sequential version.

The behaviour for grid 200×128 (Figure 5b), apart from also an increased number of iterations with the Reynolds number (as in the case of grid 100×64),

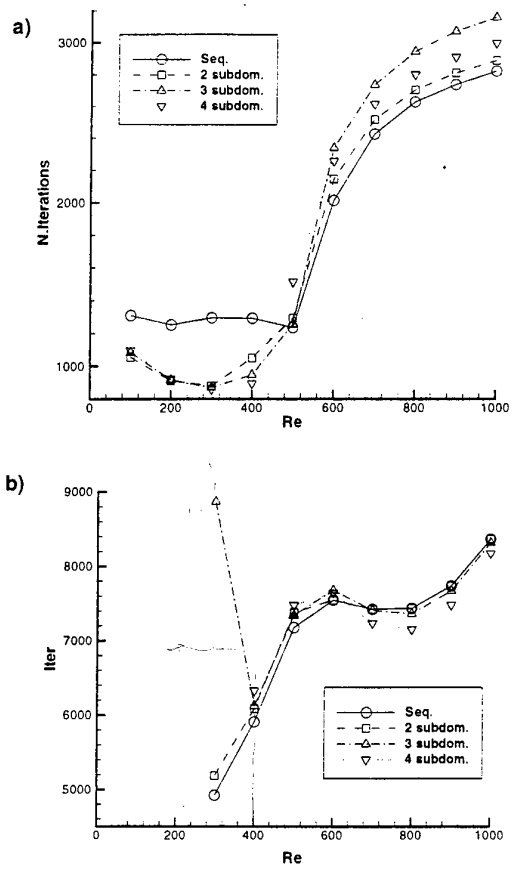


Fig. 5. Number of iterations as a function of Reynolds number and number of subdomains for two numerical grids. a) 100×64; b) 200×128.

evidences other details worth referring to. For instance, convergence could not be achieved for Reynolds number lower than 300, even in case of the sequential version. There was no optimization of the under-relaxation factors. This exercise was beyond the scope of the present work. It is our experience [12] that the SIMPLE algorithm requires lower under-relaxation factors for finer grids. The plateau shown for $500 < Re < 800$ by grid 200×128 may be associated with the occurrence of the recirculation region at the bottom wall around $x/h=17$. A feature to which the coarse grid could not be sensitive because of lack of resolution. This is a statement that we found difficult to confirm.

Figure 5 shows that the number of subdomains did not alter the general pattern of the convergence characteristics compared with the sequential version. This is valid throughout the current study and led us to the conclusion that the

domain splitting, even when the interface between the subdomains divides the recirculation region, does not affect the convergence.

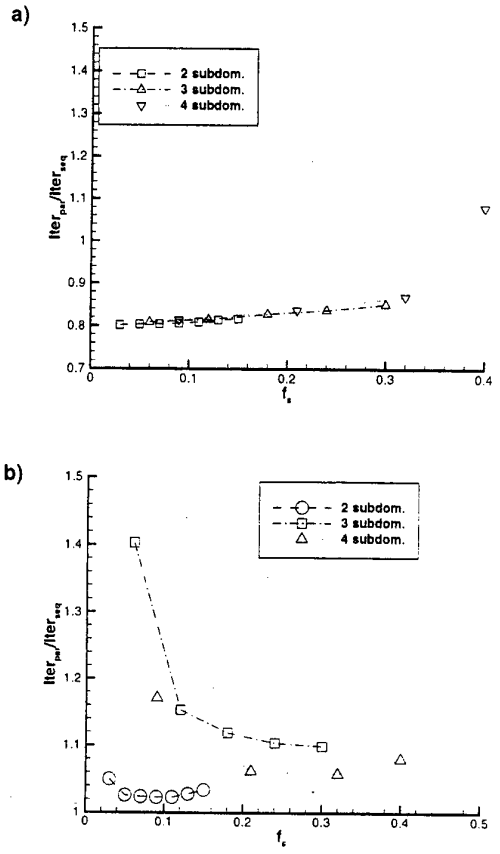


Fig. 6. Number of iterations as a function of subdomain overlapping and number of subdomains for grid 100×64 and two Reynolds number. a) Re=100; b) Re=1000.

3.3 Domain overlapping

The effect of overlapping between subdomains was studied [1] for Re=100, 500 and 1000 on the grid 100×64. Results are shown here (Figure 6) for Re=100 and 1000 as a function of f_s, defined as the ratio between the number of nodes in the overlapping regions and the total number of grid nodes. The number of nodes was identical for each subdomain, to avoid load balancing problems. This,

however, restricted the overlapping to only a few values, depending on the grid nodes and on the number of subdomains.

The results at Re=100 (Figure 6a) are markedly different from those at Re=1000 (Figure 6b). At Re=100 the parallel requires less iterations than the sequential version and is not sensitive to the number of subdomains. The number of iterations increases slightly with the overlapping, with the exception of f_s=0.4.

At Re=1000 for each case, with 2, 3 or 4 subdomains, there is a value of f_s beyond which there is no reduction of the number of iterations. Furthermore, for all cases, the number of iterations of the parallel exceeds the number of iterations of the sequential version.

3.4 Speed-up

The speed-up (S_n) was defined by

$$S_n = \frac{T_S}{T_p} \tag{3}$$

where T_S and T_p stand for the execution time of the sequential and parallel version of the code. The computing time was the actual wall-clock elapsed time, as given by the UNIX command **time** and following the recommendations of ref. [9].

One must remember that the parallel version performs extra calculations, because the grid nodes within the overlapping region are calculated twice. Taking that into account, the expected speed-up values are 1.8, 2.5 and 3.2, for 2, 3 and 4 subdomains, respectively. These values are shown by horizontal lines in Figure 7 and were obtained assuming identical number of iterations for both sequential and parallel calculations, time per iteration directly proportional to the number of nodes and negligible communication overhead.

Figure 7a shows that for grid 100×64 and Reynolds number lower than 500 the speed-up exceeded the expected value. The domain splitting was favourable to the convergence of the algorithm, and the number of iterations was reduced compared with the sequential version. For Reynolds numbers higher than 500, the speed-up was below the ideal value, in particular for cases with 3 and 4 subdomains. Nevertheless in real time, in case of 4 subdomains a converged solution can be obtained in at least 2.6× faster compared with a sequential calculation.

Figure 7b shows a fairly uniform trend and all values were close to the maximum speed-up.

4 Conclusions

Results were shown of the simulation of the laminar backward-facing step flow. A parallelized version of the SIMPLE algorithm was used, based on the partitioning of the domain. The main conclusions were:

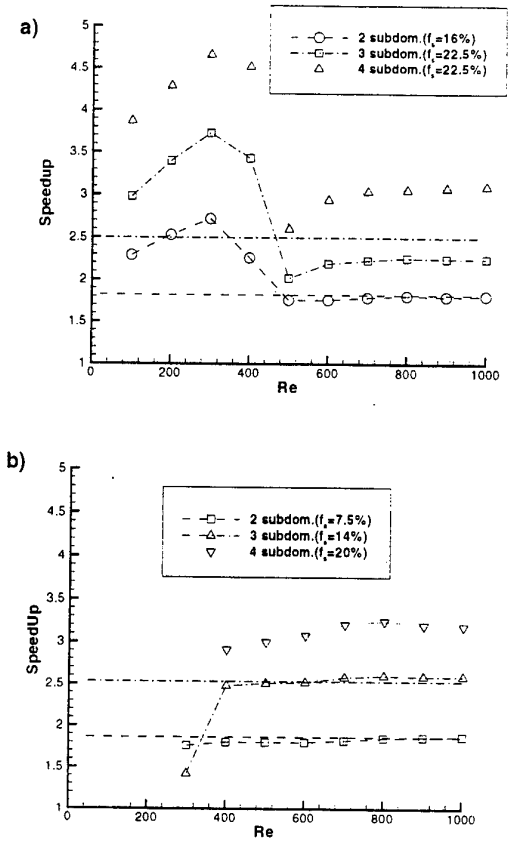


Fig. 7. Speed-up as a function of the Reynolds number, in case of 2, 3 and 4 subdomains for two numerical grids. a) 100×64; b) 200×128.

1. The minimum number of iterations was obtained for Reynolds number about 400 or 500, depending on the grid size. The convergence pattern was tightly coupled with the flow pattern, and the number of iterations increased as soon as a second recirculation region attached to the top wall appeared.
2. The comparison between the sequential and the parallel versions of the algorithm showed that the number of iterations was in many cases identical in both versions. The communication overhead was 2% of the total computing time.
3. The number of subdomains did not alter the general pattern of the convergence characteristics compared with the sequential version.
4. The convergence was not affected by domain splitting, even when the interface between the subdomains divided the recirculation region.

Acknowledgments

The authors are grateful to their colleagues F.A. Castro and A. Silva Lopes for helpful discussions.

This work was carried out as part of *PRAXIS XXI* research contract N. 3/3.1/CEG/2511/95, entitled *"Parallel Algorithms in Fluid Mechanics"*.

References

1. P. Areal. Studies of parallelization of algorithms for computational fluid dynamics (*in Portuguese*). Master's thesis, University of Porto (Faculty of Engineering), Porto, Portugal, 1998. In preparation.
2. B.F. Armaly, F. Durst, J.C.F. Pereira, and B. Schönung. Experimental and theoretical investigation of backward-facing step flow. *Journal of Fluid Mechanics*, 127:473–496, 1983.
3. L.M.R. Carvalho and J.M.L.M. Palma. *Parallelization of CFD code using PVM and domain decomposition techniques*, pages 247–257. In [13], 1997.
4. F.A. Castro. *Numerical Methods for Simulation of Atmospheric Flows over Complex Terrain (In Portuguese)*. PhD thesis, University of Porto (Faculty of Engineering), Porto, Portugal, 1997.
5. F. Dias d'Almeida, F.A. Castro, J.M.L.M. Palma, and P. Vasconcelos. Development of a parallel implicit algorithm for CFD calculations. Presented at the AGARD *77th Fluid Dynamics Panel Symposium on Progress and Challenges in CFD Methods and Algorithms*, 2–5 October 1995, Seville, Spain, 1995.
6. J.H. Ferziger and M. Perić. *Computational Methods for Fluid Dynamics*. Springer, Berlin, 1996.
7. G.A. Geist, A. Beguelin, J.J. Dongarra, W. Jiang, R. Manchek, and V.S. Sunderam. *PVM Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation. The MIT Press, Cambridge, Massachusetts, 1994.
8. N.N. Ghia, G.A. Osswald, and U. Ghia. A direct method for the solution of unsteady two-dimensional incompressible Navier-Stokes equations. *Aerodynamic Flows*, January.
9. R.W. Hockney. *The Science of Computer Benchmarking*. Software, Environments and Tools. SIAM – Society for Industrial and Applied Mathematics, Philadelphia, USA, 1996.
10. J. Kim and P. Moin. Application of a fractional step method to incompressible Navier-Stokes equations. *Journal of Computational Physics*, 59:308–, 1985.
11. M. Kurreck and S. Wittig. A comparative study of pressure correction and block-implicit finite volume algorithms on parallel computers. *International Journal for Numerical Methods in Fluids*, 24:1111–1128, 1997.
12. J.J. McGuirk and J.M.L.M. Palma. The efficiency of alternative pressure-correction formulations for incompressible turbulent flow problems. *Computers & Fluids*, 22(1):77–87, 1993.
13. J.M.L.M. Palma and J. Dongarra, editors. *Vector and Parallel Processing – VECPAR'96. Second International Conference on Vector and Parallel Processing – Systems and Applications, Porto (Portugal). Selected Papers*, volume 1215 of *Lecture Notes in Computer Science*. Springer, Berlin, 1997.

14. S.V. Patankar and D.B. Spalding. A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. *International Journal of Heat and Mass Transfer*, 15:1787–1806, 1972.
15. C.M. Rhie and W.L. Chow. Numerical study of the turbulent flow past an airfoil with trailing edge separation. *AIAA Journal*. 21(11):1525–1532, 1983.
16. R.L. Simpson. Two-dimensional turbulent separated flows. Technical report, AGARDograph N. 287, 1985.
17. H.L. Stone. Iterative solution of implicit approximations of multidimensional partial differential equations. *SIAM Journal of Numerical Analysis*, 5:530–558, 1968.
18. Y. Zang, R.L. Street, and J.R. Koseff. A non-staggered grid, fractional step method for time-dependent incompressible Navier-Stokes equations in curvilinear coordinates. *Journal of Computational Physics*, 114:18–33, 1994.

High Performance Cache Management for Parallel File Systems

F. García, J. Carretero, F. Pérez and P. de Miguel

Facultad de Informática, Universidad Politécnica de Madrid (UPM)
Campus de Montegancedo, Boadilla del Monte, 28660 Madrid, Spain
fgarcia@fi.upm.es

Abstract. Caching has been intensively used in memory and traditional file systems to improve system performance. However, the use of caching in parallel file systems has been limited to I/O nodes to avoid cache coherence problems. In this paper we present the cache mechanisms implemented in *ParFiSys*, a parallel file system developed at the UPM. *ParFiSys* exploits the use of cache, both at processing and I/O nodes, with aggressive prefetching and delayed-write techniques. The cache coherence problem is solved by using a dynamic scheme of cache coherence protocols with different sizes and shapes of granularity. Performance results, obtained on an IBM SP2, are presented to demonstrate the advantages offered by the cache management methods used in *ParFiSys*.
Keywords: Parallel file systems, data declustering, cache coherence protocols, false sharing, multi-files.

1 Introduction

There is a general trend to use *parallelism* in the I/O systems to alleviate the growing disparity in computational and I/O capability of the parallel and distributed architectures. Parallelism in the I/O subsystem is obtained using several independent I/O nodes supporting one or more secondary storage devices. Data are *declustered* among these nodes and devices to allow parallel access to different files, and parallel access to the same file. Parallelism has been used in some parallel file systems and I/O libraries described in the bibliography (CFS [20], Vesta [5], *ParFiSys* [2], PASSION [4], Galley [19], Scotch [11], PIOUS [17]).

Caching has been a technique frequently used in memory and traditional file systems to improve system performance. Caching [14, 2] can be used in parallel file systems, by allocating a buffer cache at the processing nodes (PN) and I/O nodes (ION). This approach improves I/O performance by avoiding unnecessary disk traffic, network traffic and servers load, and also by allowing prefetching and delayed-write techniques [14, 2]. However, the use of caching in parallel file systems has been limited to I/O nodes because any attempt to maintain caching at the processing nodes would require a cache coherence protocol.

In this paper we demonstrate that the use of caching at processing nodes is feasible in parallel file systems. With this aim we show the cache management

policies and mechanisms implemented in *ParFiSys*[1], a parallel file system developed at the UPM[2]. *ParFiSys* exploits the use of caching both at processing and I/O nodes. It avoids the cache coherence and false sharing problems in an efficient manner by using a dynamic scheme of cache coherence protocols with different sizes and shapes of granularity.

The rest of the paper is organized as follows. Section 2 describes *ParFiSys* main architectural features. Section 3 presents the cache management implemented in *ParFiSys*. The cache coherence problem and how *ParFiSys* solved this problem is explained in section 4. Performance measurements are shown in section 5. Finally, section 6 summarizes our conclusions.

2 *ParFiSys* Architecture

ParFiSys [2, 3] is a parallel and distributed file system developed at the UPM to provide parallel I/O services for parallel and distributed systems. To fully exploit all the parallel and distributed features of the I/O hardware, the architecture of *ParFiSys* is clearly divided in two levels: file services and block services (see figure 1.) The first level is comprised into a component named *ParClient*. It provides file services that can be obtained using two mechanisms: linked library or message passing library. The first is preferred for parallel machines [4, 13], where a single user is usually expected per processing node (PN). The second is aimed to be used in distributed systems, where several users may be requesting I/O services to the *ParClient*. Both modalities provide the users with a *ParClient* library that includes the POSIX interface and some high-performance extensions [2] . The main architectural difference between the former models, is that with the linked library approach the *ParClient* has to be present on every PN requesting I/O, while the message passing option allows the existence of remote users for a *ParClient*. This option is specifically though for distributed file systems or big scale parallel machines, and it can be used to define *groups* of users related with a single *ParClient* to increase scalability [2]. The *ParClient* translates user addresses to logical blocks establishing the connections with the *ParServer*. All communication is handled through a high performance I/O library, named *ParServer* library. This library optimizes the I/O requests and sends them to the I/O servers via message passing. It also controls the flow of data from the I/O servers to the application's address space and vice-versa. The *ParServer*, located at the input/output nodes (IONs), deal with logical block requests issued by the *ParClient*, translating them to the local secondary storage devices. Both levels intensively use caching to optimize I/O operations: the *ParClient* to avoid remote operations, and the *ParServer* to reduce accesses to devices.

ParFiSys uses a very generic *distributed partition* which allows to create several types of file systems on any kind of parallel I/O system. A distributed partition has a unique identifier, physical layout, list of sub-partitions, etc. The physical layout, represented as the tuple $(\{NODE_n\}, \{CTLR_c\}_n, \{DEV_d\}_{nc})$,

[1] http://laurel.datsi.fi.upm.es/~gp/parfisys.html

[2] *ParFiSys* has been developed under EU's ESPRIT Project GPMIMD (P-5404)

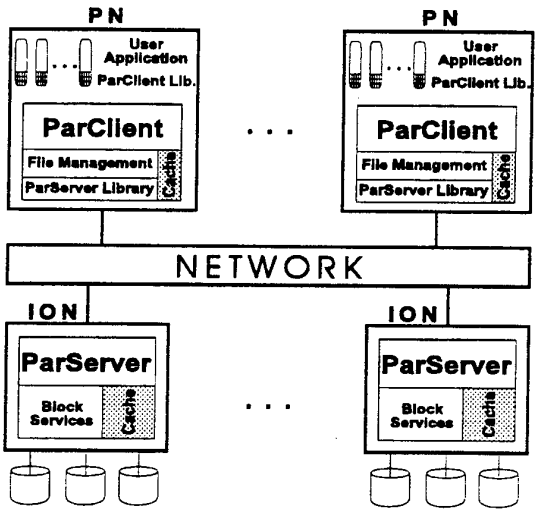


Fig. 1. *ParFiSys* Architecture

describes the set of I/O nodes, controllers per node, and devices per controller. *ParFiSys* partitions can be modified by the administrator freely, adding or removing devices dynamically. The only restriction to be considered is that devices being used by the existing applications should not be affected. The current implementation of *ParFiSys* supports three kinds of predefined file systems on the partition structure (see figure 2):

- UNIX-like non-distributed file systems, where $|NODE| = 1$, $|CTLR| = 1$, $|DEV| = 1$. .
- Extended distributed file systems with sequential layout, where $|NODE| = k$, $|CTLR| = n$, $|DEV| = m$. They can be seen as a concatenation of UNIX-like partitions.
- Distributed file systems striped with cyclic layout, where $|NODE| = k$, $|CTLR| = n$, $|DEV| = m$. Blocks are distributed through the partition devices following a *round-robin* pattern.

3 Cache Management in *ParFiSys*

ParFiSys exploits the use of caching both in *ParClient* and *ParServer* (see figure 1), allowing multiple readers and writers concurrently. Specially important is the use of cache at *ParClient*. Each *ParClient* has a buffer cache that is maintained by a *ParClient cache manager*. When a user request is received in a *ParClient*, the whole buffer is analyzed to obtain the list of file blocks associated to the buffer.

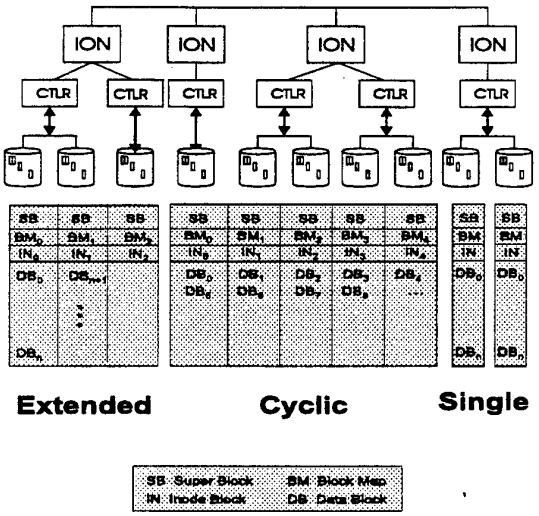


Fig. 2. File Systems Available on *ParFiSys*

Once the user buffer has been mapped to file blocks, the whole block list is searched in the *ParClient* cache with a single search operation. After this step, two new lists of blocks are obtained: *present* and *absent* blocks. The *present* blocks, those found on the cache, are immediately copied to the user space. The *absent* blocks, those not found in the cache, have to be requested to the I/O devices through the *ParServer* library. This library is conceived as a high-level device driver that concurrently manages the ION related operations. Requests not serviced are enqueued and consumed later by a thread that explores the list of blocks from the queue and generates an independent list of blocks per each ION. These blocks are requested concurrently to each *ParServer*, overlapping I/O and computation (see figure 3.) The result of this stage will be different depending on the mapping function associated to each logical device. Anyway, once the set of blocks stored into an I/O node has been determined, a thread is awaken to take care of the *ParClient-ParServer* operations related to this set of blocks. All the threads execute concurrently, notifying the end of their operations by synchronizing themselves with a barrier previously established by the *ParServer* library. An I/O operation is finished at this level only when all the threads have reached the barrier. At this moment, the result is notified to the cache management procedures and the involved I/O operations are finished. The number of client-server interactions needed depends on the maximum number of blocks, named *grouping factor*, that can be requested from each file system on a single operation.

This cache scheme exploits the parallelism in the accesses, having two main advantages: there is not synchronization among the *ParServer* involved in an

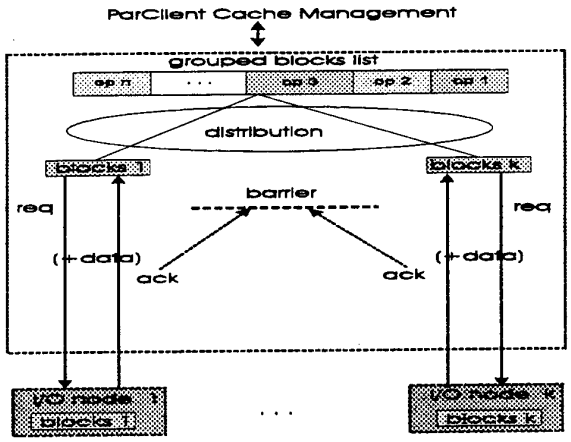


Fig. 3. Parallel Operations in *ParFiSys*

I/O operation, and there is not sequentialization in the *ParClient-ParServer* communication.

Two additional mechanisms have been used to enhance the behavior of the cache in *ParFiSys*:

Read-ahead Each *ParServer* reads ahead data using an *Infinite Block Lookahead* (IBL) predictor, that computes the number of blocks (n) to be read in advance. Prefetch is executed in the *ParClient* using an *adaptive* predictor, valid for sequential and interleaved patterns, whose behavior depends on the I/O patterns exhibited by the local processes. Prefetch is executed asynchronously to the user requests to enhance the answer time.

Write-before-full This is a *delayed-write* policy that flush dirty blocks from the *ParClient* to the *ParServer*, and from the *ParServer* to the I/O devices, before free blocks may be needed in the cache. *Preflush* is activated when a low threshold, calculated by the write-before-full daemon, is reached. When a write request is executed, the number of dirty blocks in the cache is computed. If it is larger than a fixed threshold, a massive flush is executed for the dirty blocks belonging to the file system storing the file. All the operations are executed asynchronously to the user requests to avoid delaying the answer time.

4 Avoiding the Cache Coherence Problem

The main problem of using caching at the client nodes is the possibility of having shared writing of a file from different clients [1, 18, 10], which might lead to an

incoherent view of data. Nelson [18] describes two forms of write-sharing: *sequential write-sharing* (SWS), that occurs when a client reads or writes a file that was previously written by another client, and *concurrent write-sharing* (CWS), that occurs when a file is simultaneously open for reading and writing on more than one client. Concurrent write-sharing is not usual in distributed file systems [1, 18], but it is very frequent in parallel file systems [16] and *meta-computing* [22].

A cache coherence protocol is required to avoid this problem. The use of cache coherence protocols has been unpopular in parallel file systems because of its overhead [16]. Thus, most parallel file systems usually have caching schemes such as the ones implemented in CFS [20], where only the IONs maintain a buffer cache for files. This solution avoids cache coherence problems because there only is a single copy of the data in the whole system. Distributed file systems, where write sharing is infrequent [18], use cache coherence protocols mostly based on weak [12] or coarse grain models [18]. However, most existing file systems with cache coherence protocols fail to provide efficient solutions to the problem of cache coherence for parallel applications that concurrently write a file. NFS [23], very popular in commercial environments, is unable to maintain a consistent view of the file system for parallel applications. AFS [12] does not support concurrent write-sharing due to the session semantic implemented, which makes it not suitable for parallel applications. Sprite [18] ensures concurrent write-sharing coherence by disabling client caches, thus limiting the potential benefits of caching for many parallel applications. There are very few cache coherence solutions in parallel file systems. ENWRICH [21] provides a cache coherence solution for parallel file systems, but it is not a general one because client caches are only used for writing.

Recently other approaches to avoid cache coherence problem in parallel and distributed file systems have been proposed. An example is the cooperative cache scheme proposed in [7], that eliminates the cache coherence problem by avoiding data replication. This solution, however, reduces the potential parallelism that the use of data replication may offer.

4.1 *ParFiSys* coherence model

In order to solve the write-sharing problem, *ParFiSys* uses a dynamic cache coherence scheme [9, 10] based on the following protocols:

- *Sequential coherence protocol* (SCP), that solves the SWS problem and detects the CWS on a file.
- *Concurrent coherence protocol* (CCP), that solves the CWS problem after being activated when the SCP detects a CWS situation on a file.

Sequential coherence protocol This protocol ensures coherence in SWS situations and detects CWS situations on a file. It has a behavior similar to the Sprite protocol [18]: the servers track *open* and *close* operations to know not only which clients are currently using a file, but whether any of them are potential

writers. When a client opens a file, the event is notified to the server storing the file descriptor. If there is no CWS for the file, the server looks whether another client has updated the file data in its local cache, because of the delayed-write policy, and requests it to flush the data. When the server has the most up to date copy of the file, a message is sent to the client to enable local caching for the file. No more interactions with the server are needed to maintain coherence, thus alleviating overhead. When a CWS situation is detected by the server, a message is sent to all the clients with the file open to activate the CCP. When the CWS situation disappears, a message is sent to all the clients with the file open to deactivate the CCP.

SCP has been optimized to reduce client-server interactions by sending coherence messages to the servers only when a change of the client local state of the file occurs. The local state of a file changes when it is open the first time, when it was open for read and it is open for write, when it is closed for write and it remains open for read, and when it is closed by the last user in the client. SCP has also been optimized to reduce servers load by distributing the protocol overhead among all servers. Each server executes SCP only for the files whose descriptors are stored on it, which alleviates the bottleneck of a centralized service and improves scalability.

Concurrent coherence protocol This protocol solves the CWS problem. It is activated when a CWS is detected on a file, being executed on each access to the file while the CWS situation remains. CCP is based on invalidations, directories and the existence of a exclusive write-shared copy of data.

The main problem in cache coherence protocols is the false sharing situation generated when multiple processes, belonging to the same parallel application, access the same file for writing. To alleviate false sharing problems, it would be desirable to allow the parallel applications to adjust the granularity of the protocol to their I/O patterns. *ParFiSys* ensures this by allowing the users to define *coherence regions*. A coherence region is a disjoint subset of the file used as the coherence unit. It has two main features: *size* and *shape*. The size of a region may range from the whole file to byte. The shape of a region can be defined according to the most frequent parallel access patterns: sequential and interleaved (see figure 4).

The applications can define the mapping of the regions on a file using four parameters (see figure 4):

- *Register size*, minimum unit for coherence.
- *Register stride*, width of the register groups into each segment.
- *Segment size*, number of register groups in a segment.
- *Segment stride*, distance between two segments with the same pattern.

This model of region is suitable to map very different parallel I/O patterns. Moreover, it allows to define *optimal regions*, the best suited to the I/O access pattern, to minimize coherency overhead. Optimal regions are defined on a file when:

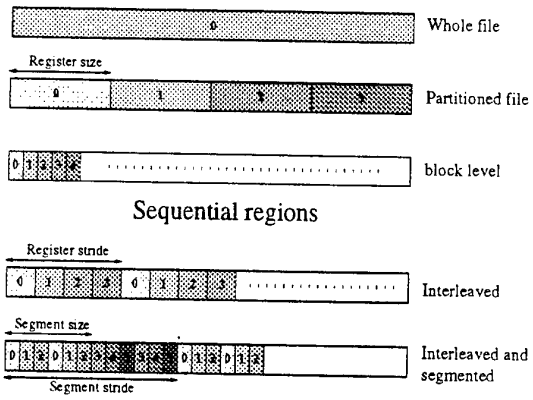


Fig. 4. Some Coherence Region Patterns

1. The number of regions is equal to the number of process in the parallel application.
2. Each process only accesses the data of a single region.

The use of optimal regions allows to adjust perfectly the protocol granularity and the I/O pattern of the applications, offering coherence with a minimal cost. This coherence regions model can be applied to the High Performance Fortran distributions [8], Vesta interface [5] and MPI-IO interface [6].

CWP compels the clients to check the coherence state of the region on every access to it by acquiring the appropriate read or write *tokens*. When a client does not have the appropriate token, a message to the region's manager must be sent to request the desired rights on the region. The server stores a *callback* for each region in a coherence directory to trace the coherence state of the region. If a conflict is detected, the callbacks are revoked. The region's manager guarantees that at any given time there is a single read-write token or any number of read-only tokens. When a write token is revoked in a client, the client must flush any dirty data of the region. If the new token is for read the token in the previous writer is changed from writing to read-only. When the appropriated rights on a region are acquired, they remain until explicit revocation, thus eliminating the overhead for future accesses. Two main design mechanisms were used in *ParFiSys* to define the coherence protocols: callback and directory location and management, and callback revocation policy. Two policies can be used for the first issue: *centralized* (C), where coherence for all the regions of a file is maintained by the server storing the file descriptor (region manager), and *distributed* (D), where the information of the coherence region is distributed among several servers, each one being responsible of maintaining the coherence of the regions allocated to it. Two approaches can be used to revoke callbacks when a conflict appears: *server driven* (SD) and *client driven* (CD). In SD, the server

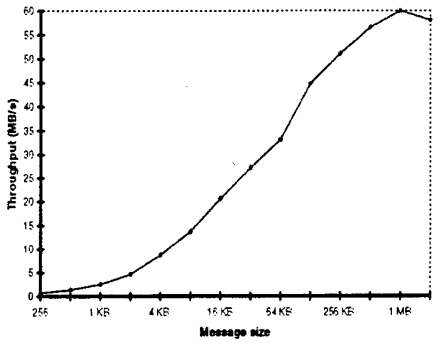


Fig. 5. Point-to-point communication throughput

sends revocation messages to all the clients caching data from the conflictive region. In CD, the client generating the conflict sends revocation messages, on behalf of the servers, to other clients caching data from the conflictive region. Several CCP have implemented in *ParFiSys* by combining different management and revocation policies: C-SD, C-CD, D-SD, and D-CD.

5 Performance Evaluation

This section describes the performance experiments designed to test cache management in *ParFiSys*a and the results obtained by running them on an IBM SP2 machine.

The IBM SP2 used is a distributed-memory MIMD machine with 14 nodes available to execute parallel applications. Each node has a 66 MHz POWER2 RISC System/6000 processor with 256 MB of memory, being connected to both an Ethernet and IBM's high performance switch. Because of the IBM's message-passing libraries (PVM, MPL or MPI) cannot operate in a multi-threaded environment, we have implemented a multi-threaded subset of MPI using TCP/IP on top of the high performance switch. To characterize the message-passing performance of our communication library, we executed two simple benchmarks on the SP2. The first evaluates point-to-point communication throughput by engaging two processes in a sort of ping-pong. One process reads the value of a wall-time clock before invoking a send operation and it then blocks in a receive operation. Once the latter operation finishes, the clock is read again. The throughput achieved is computed with a half of the communication time and the message size. Figure 5 shows the results obtained for this benchmark. The maximum throughput between two nodes is approximately 60 MB/s.

The second benchmark emulates *ParFiSys* reading and writing activity. We used 4 servers and varied the number of clients from 1 to 8. Clients send (write to servers) or receive (read from servers) 100 MB declustered across the servers. Each client sends, or receives, the same amount of data to, or from, the servers

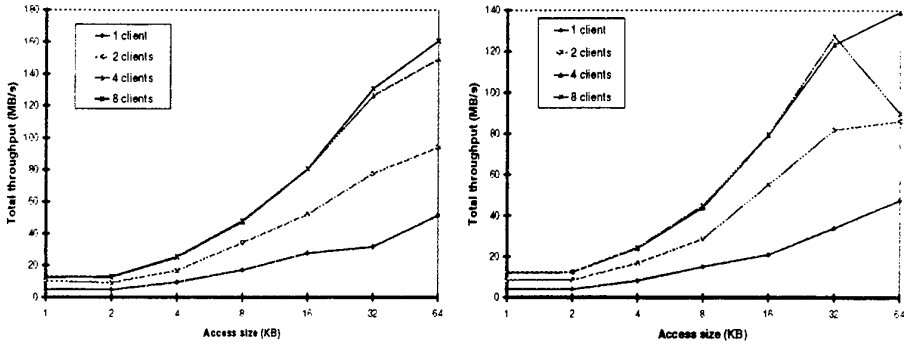


Fig. 6. Total message passing throughput in *ParFiSys*

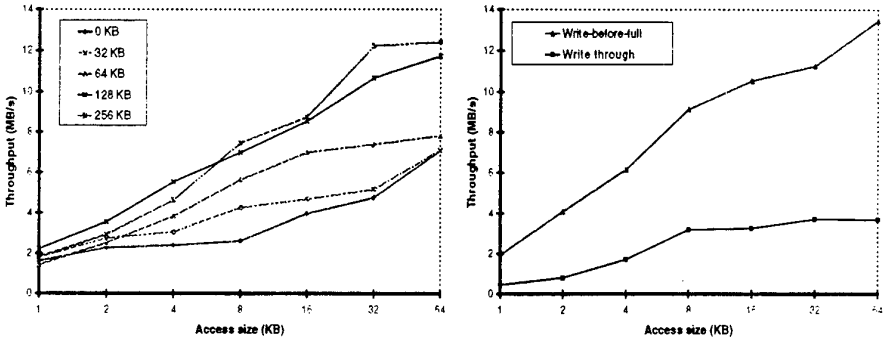


Fig. 7. Prefetching and write-before-full performance

using a fixed record size. Figure 6 shows the total throughput obtained for reading and writing operations. As shown in the figure, the maximum throughput achieved increases with the number of clients and the record size.

All experiments described below were executed using 4 ION, with a single simulated disk with a bandwidth of 5 MB/s on each. In all tests a 4 MB per-client cache and a 16 MB per-ION cache were used. The file size used in all experiments was 100 MB. The stripe-width was 4 and the stripe-unit size was 8 KB.

5.1 Prefetch and write-before-full evaluation

Figure 7, left, shows the throughput obtained when a client sequentially reads a file of 100 MB. This test varied the read-ahead value, from 0 KB to 256 KB, and the access size, from 1 KB to 64 KB.

Figure 7, right, shows the throughput obtained when a client sequentially writes a file of 100 MB using either write-through or write-before-full. The threshold used in write-before-full was a 95 % of the *ParClient* cache size. The

results obtained demonstrate that having a cache at processing nodes, managed using prefetch and write-before-full, is a useful mechanism to increase read and write performance in parallel file systems.

5.2 Cache Coherence Performance in *ParFiSys*

Two benchmarks were defined to evaluate the *ParFiSys* coherence protocol and to demonstrate their feasibility for parallel applications: a segmented concurrent write benchmark (SCWB) and a interleaved concurrent write benchmark (ICWB). The SCWB, similarly to the one described in [17], is a parallel program with partitioned access that divides a file into contiguous segments, one per process, with each segment accessed sequentially by a different process. In the ICWB, each process concurrently writes the file in a interleaved fashion. The parallel program for each benchmark consists of 1, 2, 4 and 8 processes that concurrently write a file of 100 MB. Two access sizes are used: 8 KB and 32 KB.

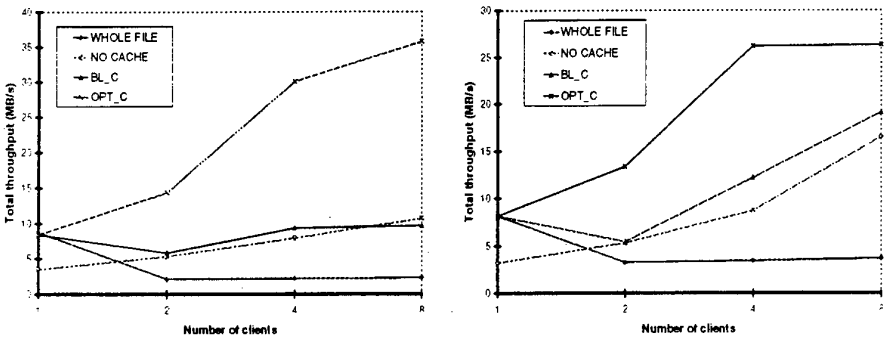


Fig. 8. Concurrent segmented and interleaved benchmark for 8 KB access size

Figures 8 and 9 show the aggregated bandwidth for concurrent segmented and interleaved I/O patterns, respectively. Several cache coherence protocols using SCWB and ICWB are evaluated: client cache deactivated (NO CACHE), file granularity (WHOLE F.), block granularity for centralized protocol (BL_C), and optimal regions for centralized protocol (OPT_C). The first relevant result obtained shows that maintaining coherency with file granularity is the worst method, mainly due to false sharing. Deactivating cache is very similar to the block centralized, because the number of client-server interactions is almost the same. The small difference observed between deactivating cache and block distributed is mainly due to the lower contention generated on the servers by the coherence protocols. This feature also makes the block distributed protocol more scalable. The best results are obtained using optimal regions, both centralized and distributed. This behavior is mainly due to the false sharing elimination,

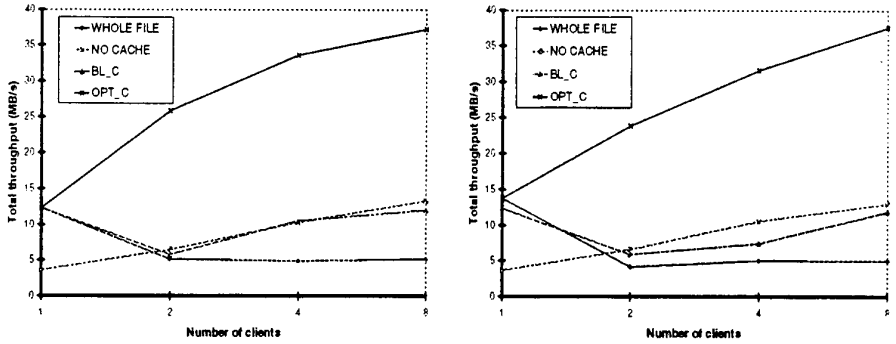


Fig. 9. Concurrent segmented and interleaved benchmark for 32 KB access size

the main problem with whole file granularity, the minimization of the coherence load, the main problem in block granularity protocols, and the local cache utilization, the main problem in cache deactivation.

Concurrent write benchmarks for segmented and interleaved I/O patterns show that optimal regions provide a performance very close to the ideal one. Moreover, they show a good scalability compared with the other protocols, whose performance decreases very quickly as the number of clients is increased.

Figure 10 compares the bandwidth obtained in the SCWB using normal files with optimal regions for centralized and distributed protocols, versus the use of multi-files. A multi-file [15, 2] is a collection of subfiles, each of which is a separate sequence of bytes. A multi-file is created by a parallel program with a certain number of subfiles, usually equal to the number of processes in the program, with each process accessing its own subfile. A multi-file combines the advantages of a single file, single name for a single data set, with those of multi-files, that are independently addressable. Multi-files are an alternative mechanism to the segmented patterns used in SCWB. The results are very similar in all cases, being a little better for multi-files because of the lighter writing operation on each subfile than on a normal file with concurrent access. The proposed protocols with normal files offer an efficient alternative to the use of multi-files for segmented I/O patterns, because present a performance very similar with the advantage of using less specialized and more portable interfaces.

6 Conclusions

This paper has presented the design of the cache scheme implemented in *ParFiSys*. *ParFiSys* allocates buffer caches at the processing and I/O nodes, improving I/O performance by avoiding unnecessary disk traffic, network traffic and servers load, and by allowing prefetching and delayed-write techniques. The cache coherence problem is solved, without loss of scalability, by using a dynamic scheme of cache coherence protocols where data coherence is maintained on user-defined

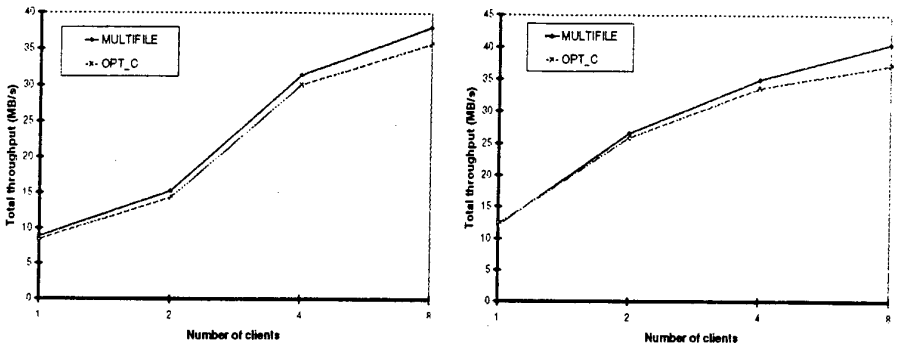


Fig. 10. Optimal regions in SCWB versus multi-files for 8K B and 32 KB access size

coherence regions for the conflictive file. The utilization of two protocols, SWP and CWP, allows to afford all the conflictive situations for SWS and CWS patterns, as demonstrated with the evaluation results obtained by running *ParFiSys* on an IBM SP2. The benchmarks used to test the model show considerably better results for our model than for other existing models. The aggregated bandwidth obtained is higher when using our model, mainly because false sharing is reduced, coherence load is minimized, and local caches at processing nodes are heavily used. The proposed protocols also offer an efficient alternative to the use of multi-files for segmented I/O patterns, because present a very similar performance with the advantage of using less specialized and more portable interfaces.

Acknowledgments We want to express our grateful acknowledgment to the CESCA institution for giving us access to their IBM SP2 machine.

References

1. M. Burrows. *Efficient Data Sharing.* PhD thesis, Computer Laboratory, University of Cambridge, December 1988.
2. J. Carretero. *Un Sistema de Ficheros Paralelo con Coherencia de Cache para Multiprocesadores de Proposito General.* PhD thesis, Universidad Politécnica de Madrid, May 1995.
3. J. Carretero, F. Pérez, P. De Miguel, F. García, and L. Alonso. Performance Increase Mechanisms for Parallel and Distributed File Systems. *Parallel Computing.* (23):525–542, August 1997.
4. Alok Choudhary, Rajesh Bordawekar, Sachin More, K. Sivaram, and Rajeev Thakur. PASSION runtime library for the Intel Paragon. In *Proceedings of the Intel Supercomputer User's Group Conference*, June 1995.
5. P. Corbett and D. Feitelson. The Vesta Parallel File System. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.
6. P. Corbett and D. et al. Feitelson. MPI-IO: A Parallel File I/O Interface for MPI. Technical Report NAS-95-002, NASA Ames Research Center, June 1995.

7. T. Cortes, S. Girona, and J Labarta. Dessign Issues of a Cooperative Cache with no Coherence Problems. In *5th Workshop on I/O in Parallel and Distributed Systems (IOPADS'97), San Jose, CA, November*, 1997.
8. High Performance Fortran Forum. *High Performance Fortran Language Specification.* May 1993.
9. F. García. *Coherencia de Cache en Sistemas de Ficheros para Entornos Distribuidos y Paralelos.* PhD thesis, Universidad Politécnica de Madrid, España, September 1996.
10. F. García, J. Carretero, F. Pérez, P. De Miguel, and L. Alonso. Cache Coherence in Parallel and Distributed File Systems. In *5th EUROMICRO Workshop on Parallel and Distributed Processing. IEEE. London*, pages 60–65, January 1997.
11. G. Gibson. The Scotch Paralell Storage Systems. Technical Report CMU-CS-95-107, Scholl of Computer Science, Carnegie Mellon University, Pittsburbh, Pennsylvania, 1995.
12. J. Howard and et al. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
13. J. Huber and C. L. et al. Elford. PPFS: A High Performance Portable Parallel File System. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, July 1995.
14. D. Kotz. *Prefetching and Caching Techniques in File Systems for MIMD Multiprocessors.* PhD thesis, Duke University, USA, April 1991.
15. D. Kotz. Multiprocessor file system interfaces. In *Proceedings of the 2nd. International Conference on Parallel and Distributed Information Systems*, pages 194–201, May 1993.
16. D. Kotz and N. Nieuwejaar. File System Workload on a Scientific Multiprocessor. *IEEE Parallel and Distributed Technology. Systems and Aplications*, pages 134–154, Spring 1995.
17. S. A. Moyer and V. S. Sunderam. Scalable concurrency control for parallel file systems. Technical Report CSTR-950202, Department of Math and Computer Science, Emory University, Atlanta, GA 30322, USA, 1995.
18. M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.
19. N. Nieuwejaar and D. Kotz. The Galley Parallel File System. Technical Report PCS-TR96-286, Darmouth College Computer Science, 1996.
20. J. Pieper. Parallel I/O Systems for Multicomputers. Technical Report CMU-CS-89-143, Carneghie Mellon University, Computer Science Department, Pittsburgh, USA, 1989.
21. A. Purakayastha, C. S. Ellis, and D. Kotz. ENWRICH: A Computer-Processor Write Caching Scheme for Parallel File Systems. Technical Report TRCS-1995-22, Department of Computer Science, Duke University, Durhan North Carolina, October 1995.
22. J. del Rosario and A. N. Choudhary. High-Performance I/O for Massively Parallel Computer s. *IEEE Computer*, pages 59–68, March 1994.
23. R. Sandberg, D. Goldberg, S. Kleiman, D Walsh, and B. Lyon. Design and Implementation of the SUN Network Filesystem. In *Proc. of the 1985 USENIX Conference.* USENIX, 1985.

Parallel Jacobi-Davidson for Solving Generalized Eigenvalue Problems

Margreet Nool[†] and Auke van der Ploeg[‡]

[†] CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Margreet.Nool@cwi.nl
[‡] MARIN, P.O. Box 28, 6700 AA Wageningen, The Netherlands
A.v.d.Ploeg@marin.nl

Abstract. We study the Jacobi-Davidson method for the solution of large generalized eigenproblems as they arise in MagnetoHydroDynamics. We have combined Jacobi-Davidson (using standard Ritz values) with a shift and invert technique. We apply a complete LU decomposition in which reordering strategies based on a combination of block cyclic reduction and domain decomposition result in a well-parallelizable algorithm. Moreover, we describe a variant of Jacobi-Davidson in which harmonic Ritz values are used. In this variant the same parallel LU decomposition is used, but this time as a preconditioner to solve the 'correction' equation.
The size of the relatively small projected eigenproblems which have to be solved in the Jacobi-Davidson method is controlled by several parameters. The influence of these parameters on both the parallel performance and convergence behaviour will be studied. Numerical results of Jacobi-Davidson obtained with standard and harmonic Ritz values will be shown. Executions have been performed on a Cray T3E.

1 Introduction

Consider the generalized eigenvalue problem

$$Ax = \lambda Bx, \qquad A, B \in \mathcal{C}^{N_t \times N_t}, \tag{1}$$

in which A and B are complex block tridiagonal N_t-by-N_t matrices and B is Hermitian positive definite. The number of diagonal blocks is denoted by N and the blocks are n-by-n, so $N_t = N \times n$. In close cooperation with the FOM Institute for Plasma Physics "Rijnhuizen" in Nieuwegein, where one is interested in such generalized eigenvalue problems, we have developed a parallel code to solve (1). In particular, the physicists like to have accurate approximations of certain interior eigenvalues, called the *Alfvén spectrum*. A promising method for computing these eigenvalues is the Jacobi-Davidson (JD) method [3, 4]. With this method it is possible to find several interior eigenvalues in the neighbourhood of a given target σ and their associated eigenvectors.

In general, the subblocks of A are dense, those of B are rather sparse ($\approx 20\%$ nonzero elements) and N_t can be very large (realistic values are $N = 500$ and

$n = 800$), so computer storage demands are very high. Therefore, we study the feasibility of parallel computers with a large distributed memory for solving (1).

In [2], Jacobi-Davidson has been combined with a parallel method to compute the action of the inverse of the block tridiagonal matrix $A - \sigma B$. In this approach, called DDCR, a block-reordering based on a combination of Domain Decomposition and Cyclic Reduction is combined with a complete block LU decomposition of $A - \sigma B$. Due to the special construction of L and U, the solution process parallelizes well.

In this paper we describe two Jacobi-Davidson variants, one using standard Ritz values and one harmonic Ritz values. The first variant uses DDCR to transform the generalized eigenvalue problem into a standard eigenvalue problem. In the second one DDCR has been applied as a preconditioner to solve approximately the 'correction' equation. This approach results also into a projected standard eigenvalue problem with eigenvalues in the dominant part of the spectrum. In Section 2 both approaches are described. To avoid that the projected system becomes too large, we make use of a restarting technique. Numerical results, based on this technique, are analyzed in Section 3. We end up with some conclusions and remarks in Section 4.

2 Parallel Jacobi-Davidson

2.1 Standard Ritz values

The availability of a complete LU decomposition of the matrix $A - \sigma B$ gives us the opportunity to apply Jacobi-Davidson to a *standard* eigenvalue problem instead of a *generalized* eigenvalue problem. To that end, we rewrite (1) as

$$(A - \sigma B)x = (\lambda - \sigma)Bx. \tag{2}$$

If we define $Q := (A - \sigma B)^{-1}B$ then (2) can be written as

$$Qx = \mu x, \quad \text{with } \mu = \frac{1}{\lambda - \sigma} \Leftrightarrow \lambda = \sigma + \frac{1}{\mu}. \tag{3}$$

The eigenvalues we are interested in form the dominant part of the spectrum of Q, which makes them relatively easy to find. The action of the operator Q consists of a matrix-vector multiplication with B, a perfectly scalable parallel operation, combined with two triangular solves with L and U.

At the k-th step of Jacobi-Davidson, an eigenvector x is approximated by a linear combination of k search vectors v_j, $j = 1, 2, \cdots, k$, where k is very small compared with N_t. Consider the N_t-by-k matrix V_k, whose columns are given by v_j. The approximation to the eigenvector can be written as $V_k s$, for some k-vector s. The search directions v_j are made orthonormal to each other, using Modified Gram-Schmidt (MGS), hence $V_k^* V_k = I$.

Let θ denote an approximation of an eigenvalue associated with the Ritz vector $u = V_k s$. The vector s and the scalar θ are constructed in such a way that

the residual vector $r = QV_k s - \theta V_k s$ is orthogonal to the k search directions. From this Rayleigh-Ritz requirement it follows that

$$V_k^* Q V_k s = \theta V_k^* V_k s \Longleftrightarrow V_k^* Q V_k s = \theta s. \tag{4}$$

The size of the matrix $V_k^* Q V_k$ is k. By using a proper restart technique k stays so small that this 'projected' eigenvalue problem can be solved by a sequential method.

In order to obtain a new search direction, Jacobi-Davidson requires the solution of a system of linear equations, called the 'correction equation'. Numerical experiments show that fast convergence to selected eigenvalues can be obtained by solving the correction equation to some *modest accuracy* only, by some steps of an inner iterative method, e.g. GMRES.

Below we show the Jacobi-Davidson steps used for computing several eigenpairs of (3) using standard Ritz values.

step 0: initialize
Choose an initial vector v_1 with $\|v_1\|_2 = 1$; set $V_1 = [v_1]$;
$W_1 = [Qv_1]$; $k = 1$; $it = 1$; $n_{ev} = 0$
step 1: update the projected system
Compute the last column and row of $H_k := V_k^* W_k$
step 2: solve and choose approximate eigensolution of projected system
Compute the eigenvalues $\theta_1, \cdots, \theta_k$ of H_k and choose $\theta := \theta_j$ with $|\theta_j|$ maximal and $\theta_j \neq \mu_i$, for $i = 1, \cdots, n_{ev}$; compute associated eigenvector s with $\|s\|_2 = 1$
step 3: compute Ritz vector and check accuracy
Let u be the Ritz vector $V_k s$; compute the residual vector $r := W_k s - \theta u$;
`if` $\|r\|_2 < tol_{sJD}.|\theta|$ `then`
$n_{ev} := n_{ev} + 1$; $\mu_{n_{ev}} := \theta$; `if` $n_{ev} = N_{ev}$ `stop`; `goto 2`
`else if` $it = iter$ `stop`
`end if`
step 4: solve correction equation approximately with it_{SOL} steps of GMRES
Determine an approximate solution $\tilde{z}$ of z in
$(I - uu^*)(Q - \theta I)(I - uu^*)z = -r \ \wedge \ u^* z = 0$
step 5: restart if projected system has reached its maximum order
`if` $k = m$ `then`
5a: Set $k = k_{min} + n_{ev}$. Construct $C \in \mathcal{C}^{m \times k} \subset H_m$;
Orthonormalize columns of C; compute $H_k := C^* H_m C$
5b: Compute $V_k := V_m C$; $W_k := W_m C$
`end if`
step 6: add new search direction
$k := k + 1$; $it := it + 1$; call $MGS\,[V_{k-1}, \tilde{z}]$; set $V_k = [V_{k-1}, \tilde{z}]$; $W_k = [W_{k-1}, Q\tilde{z}]$;
`goto 1`

Steps 2 and 5a deal with the small projected system (4). Those sequential steps are performed by all processors in order to avoid communication. The basic ingredients of the other steps are matrix-vector products, vector updates and

inner products. Since, for our applications, N_t is much larger than the number of processors, those steps parallelize well.

2.2 Harmonic Ritz values

For the introduction of harmonic Ritz values we return to the original generalized eigenvalue problem (1). Assume $(\theta, V_k s)$ approximates an eigenpair (λ, x), then the residual vector r is given by

$$r = AV_k s - \theta B V_k s.$$

In case of standard Ritz values, the correction vector r has to be orthogonal to V_k; the harmonic Ritz values approach asks for vectors r to be orthogonal to $(A - \sigma B)V_k$. Let W_k denote $(A - \sigma B)V_k$, then we have

$$\begin{aligned} r &= AV_k s - \theta B V_k s \\ &= (A - \sigma B)V_k s - (\theta - \sigma)B(A - \sigma B)^{-1}(A - \sigma B)V_k s \\ &= W_k s - (\theta - \sigma)B(A - \sigma B)^{-1} W_k s. \end{aligned} \tag{5}$$

Obviously, $\nu = \frac{1}{(\theta - \sigma)}$ is a Ritz value of the matrix $B(A - \sigma B)^{-1}$ with respect to W_k. To obtain eigenvalues in the neighborhood of σ, ν must lie in the dominant spectrum of $B(A - \sigma B)^{-1}$. The orthogonalization requirement leads to

$$\nu {W_k}^* W_k s = {W_k}^* B V_k s. \tag{6}$$

To obtain a standard eigenvalue problem we require ${W_k}^* W_k = I$. By introducing $C := (A - \sigma B)^*(A - \sigma B)$ this requirement gives

$${W_k}^* W_k = {V_k}^*(A - \sigma B)^*(A - \sigma B)V_k = {V_k}^* C V_k = I \tag{7}$$

and we call V_k a C-orthonormal matrix.

The new search direction $\tilde{v}_k$ must be C-orthonormal to V_{k-1}, which implies that

$${V_{k-1}}^* v_k = 0 \text{ and } \tilde{v}_k = \frac{v_k}{\|v_k\|_C} = \frac{v_k}{\|w_k\|_2}, \tag{8}$$

where $w_k = (A - \sigma B)v_k$.

To move from standard to harmonic Ritz values, the adjustments in the algorithm are not radical. In comparison to the original implementation, the harmonic case requires two extra matrix-vector multiplications and in addition extra memory to store an N_t-by-k matrix. The main difference is that the LU decomposition of $A - \sigma B$ is used as a preconditioner and not as a shift and invert technique.

3 Numerical results

In this section, we show some results obtained on both an 80 processor Cray T3E situated at the HPαC centre in Delft, The Netherlands and a 512 processor Cray T3E at Cray Research, Eagan, MN, USA. The local memory per processor is at least 128 Mbytes. On these machines, the best results were obtained by a `MESSAGE PASSING` implementation using Cray intrinsic `SHMEM` routines for data transfer and communication. For more details, we refer to [2].

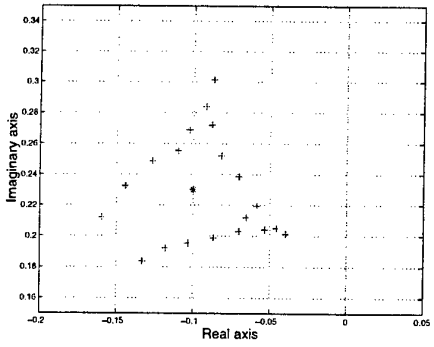


Fig. 1. The eigenvalue distribution of problem **5**

3.1 Problems

We have timed five MHD problems of the form (1). The *Alfvén* spectra of Problems **1, 2** and **3**, on the one hand, and Problems **4** and **5**, on the other hand, do not correspond because different MHD equilibria have been used. For more details we refer to CASTOR [1]. The choices of the acceptance criteria will be explained in the next section.

1 A small problem of $N = 64$ diagonal blocks of size $n = 48$. We look for eigenvalues in the neighbourhood of $\sigma = (-0.08, 0.60)$, and stop after 10 eigenpairs have been found with $tol_{sJD} = 10^{-8}$ and $tol_{hJD} = 10^{-6}$. The experiments have been performed on $p = 8$ processors.

2 The size of this problem is four times as big as that of the previous problem; $N = 128$ and $n = 96$. Again, we look for eigenvalues in the neighbourhood of $\sigma = (-0.08, 0.60)$, and stop after 10 eigenpairs have been found with $tol_{sJD} = 10^{-8}$ and $tol_{hJD} = 10^{-6}$. The experiments have been performed on $p = 8$ processors.

3 The same as Problem **2**, but performed on $p = 32$ processors.

4 The size of this large problem is: $N = 256$ and $n = 256$. We took $\sigma = (-0.15, .15)$ and look for $N_{ev} = 12$ eigenpairs with $tol_{sJD} = 10^{-8}$ and $tol_{hJD} = 10^{-5}$. The experiments are performed on $p = 128$ processors.

5 The size of this very large problem is: $N = 4096$ and $n = 64$, we took $\sigma = (-0.10, .23)$ leading to another branch in the *Alfvén spectrum*. Now, we look for $N_{ev} = 20$ eigenpairs with $tol_{sJD} = 10^{-8}$ and $\tilde{tol}_{hJD} = 10^{-5}$. For this problem a slightly different acceptance criterion has been applied:

$$\|r\|_2 < \tilde{tol}_{hJD}.|\sigma + \frac{1}{\nu}|.\|u\|_2. \tag{9}$$

For the harmonic case, the 2-norm of u can be very large, about 10^6, so the results can be compared with $tol_{hJD} = 10^{-6}$. At present, we prefer to control the residue as described in Section 3.2. Figure 1 shows the distribution of 20 eigenvalues in the neighborhood of $\sigma = (-0.10, .23)$.

3.2 Acceptance criterion

For the standard approach we accept an eigenpair $(\sigma+\frac{1}{\nu}, u)$ if the residual vector satisfies:

$$\|r\|_2 = \|(Q-\nu I)u\|_2 < tol_{sJD}.|\nu|, \text{ with } \|u\|_2 = 1 \tag{10}$$

and for the harmonic approach we require:

$$\|r\|_2 = \|(A-(\sigma+\frac{1}{\nu}))Bu\|_2 < tol_{hJD}.|\sigma+\frac{1}{\nu}|, \text{ with } \|u\|_C = 1. \tag{11}$$

To compare both eigenvalue solvers it is not advisable to choose the tolerance parameters tol_{sJD} equal to tol_{hJD} in (10) and (11), respectively. There are two reasons to take different values: firstly, within the same number of iterations the standard approach will result into more eigenpair solutions that satisfy (10) than into solutions that satisfy (11). Secondly, if we compute for each *accepted* eigenpair (λ, u) the true normalized residue γ defined by

$$\gamma := \frac{\|(A-\lambda B)u\|_2}{|\lambda|.\|u\|_2}, \tag{12}$$

then we see that the harmonic approach leads to much smaller γ values.

In Figure 2, the convergence behaviour of both the standard and harmonic approach is displayed, with and without restarts. A $\circ$ indicates that the eigenpair satisfies (10) or (11), a $\times$ denotes the γ value. We observe that the accuracy for the eigenpairs achieved by means of harmonic Ritz values is better than suggested by tol_{hJD}. On the other hand, tol_{sJD} seems to be too optimistic about the accuracy compared to the γ values shown in Figure 2. In our experiments we took $tol_{sJD} = 10^{-8}$ and $tol_{hJD} = 10^{-6}$ and $tol_{hJD} = 10^{-5}$. It is not yet clear to us how these parameters depend on the problem size or the choice of the target.

3.3 Restarting strategy

The algorithm has two parameters that control the size of the projected system: k_{min} and m. During each restart, the k_{min} eigenvalues with maximal norm and not included in the set of accepted eigenvalues, that correspond to the k_{min} most promising search directions are maintained. Moreover, since an implicit deflation technique is applied in our implementation, the n_{ev} eigenpairs found so far are kept in the system too. The maximum size m should be larger than $k_{min} + N_{ev}$, where N_{ev} denotes the number of eigenvalues we are looking for. The influence of several (k_{min}, m) parameter combinations on both the parallel performance and convergence behaviour is studied.

3.4 Timing results of (k_{min}, m) parameter combinations

For each experiment we take m constant and for k_{min} we choose the values $5, 10, \cdots, m - N_{ev}$. In Figures 4, 5, 6 and 7, the results of a single m value have been connected by a dashed or dotted line. Experiments with several m values

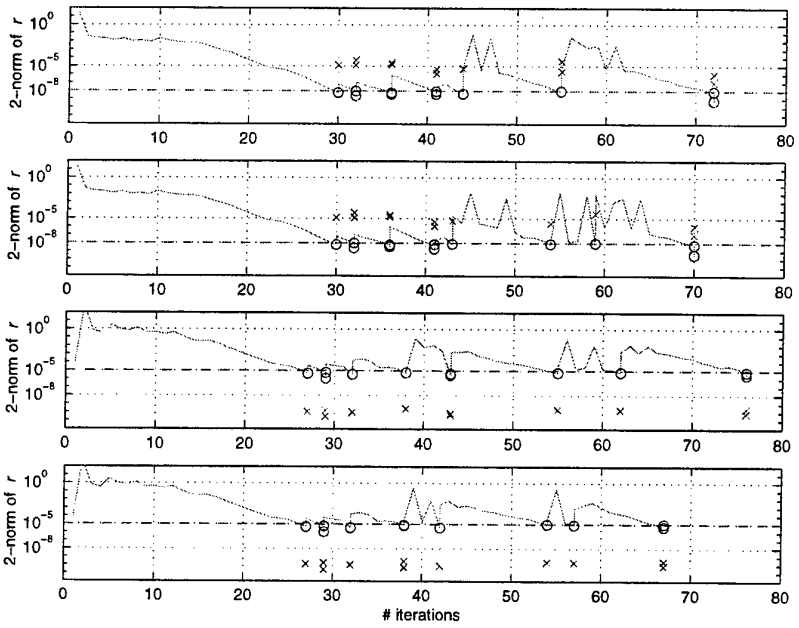


Fig. 2. The two upper plots result on problem **4** using standard Ritz values, the lower two on the same problem but using harmonic Ritz values. The first and third one show the convergence behaviour of Jacobi-Davidson restarting each time when the size of the projected system reaches $m = 37$, where $k_{min} = 25$ and $k_{min} = 20$, respectively. The second and fourth plots demonstrate the convergence in case of no restarts. The process ended when $N_{ev} = 12$ eigenvalues were found. It may happen that two eigenvalues are found within the same iteration step.

have been performed. In the plots we only show the most interesting m values; m reaches its maximum if N_{ev} eigenpairs were found without using a restart. In the pictures this is indicated by a solid horizontal line, which is of course independent of k_{min}. If the number of iterations equals 80 and besides less than N_{ev} eigenpairs have been found, we consider the result as negative. This implies that, although the execution time is low, this experiment cannot be a candidate for the best (k_{min}, m) combination.

Before we describe the experiments illustrated by Figures 4, 5, 6 and 7 we make some general remarks:

- We observed that if a (k_{min}, m) parameter combination is optimal on p processors, it is optimal on q processors too, with $p \neq q$.
- For k_{min} small, for instance $k_{min} = 5$ or 10, probably too much information is thrown away, leading to a considerable increase of iteration steps.
- For k_{min} large the number of restarts will be large at the end of the process; suppose that in the extreme case, $k_{min} = m - N_{ev}$, already $N_{ev} - 1$ eigenpairs have been found, then after a restart k becomes $k_{min} + N_{ev} - 1 = m - 1$. In other words, each step will require a restart. In Figure 3, the number of

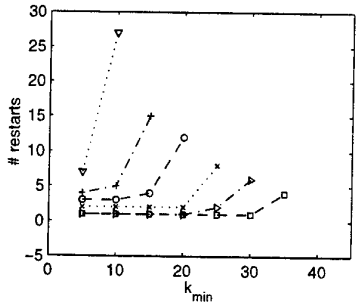


Fig. 3. The number of restarts needed to compute N_{ev} eigenvalues of Problem **2**. Results are shown for different m values: $m = 20$ ($\nabla \cdots$), $m = 25$ ($+ - \cdot$ line), $m = 30$ ($\circ - -$ line), $m = 35$ ($\times \cdots$ line), $m = 40$ ($\triangleright - \cdot$ line), $m = 45$ ($\square - -$ line).

restarts is displayed corresponding to the results of Problem **2** obtained with harmonic Ritz values.

- The number of iterations is almost independent of the number of processors involved; it may happen that an increase of the number of processors causes a decrease by one or two iterations under the same conditions, because the LU decomposition becomes more accurate if the number of cyclic reduction steps increases at the cost of the domain decomposition part.

The first example (Figure 4) explicitly shows that the restarting technique can help to reduce the wall clock time for both the standard and harmonic method. The minimum number of iterations to compute 10 eigenvalues in the neighborhood of σ is achieved in case of no restarts, viz, 53 for the standard case, 51 for the harmonic case. The least time to compute 10 eigenvalues is attained for $k_{min} = 15$ and $m = 30, 35$, but also for $k_{min} = 10$ and $m = 30, 35$ and $m = 40$ and $k_{min} = 15, 20, 25$ leads to a reduction in wall clock time of about 15 %. The harmonic approach leads to comparable results: for $(k_{min}, m) = (15, 30 : 35)$, but also $(k_{min}, m) = (10, 30 : 35)$ and $(k_{min}, m) = (15 : 25, 40)$ a reasonable reduction in time is achieved. The score for $k_{min} = 5$ in combination with $m = 35$ is striking, the unexpected small number of iterations in combination with a small k_{min} results into a fast time.

The plots in Figure 5 with the timing results for the Jacobi-Davidson process for Problem **2** give a totally different view. There is no doubt of benefit from restarting, although the numbers of iterations pretty well correspond with those of Problem **1**. This can be explained as follows: the size of the projected system k is proportionally much smaller compared to N_t/p than in case of Problem **1**; both the block size and the number of diagonal blocks is twice as big. For Problem **1** the sequential part amounts 45% and 36% of the total wall clock time, respectively, for the standard and harmonic Ritz values. For Problem **2** these values are 10.5% and 8%, respectively. These percentages hold for the most expensive sequential case of no restarts. The increase of JD iterations due to several restarts can not be compensated by a reduction of serial time by keeping

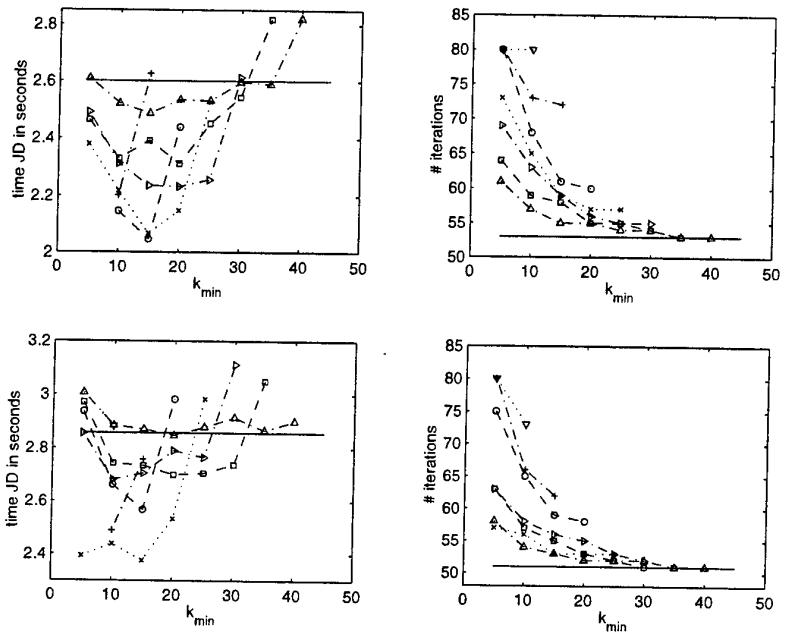


Fig. 4. The upper pictures result on problem **1** using standard Ritz values. The lower pictures result on the same problem with harmonic Ritz values. Results are shown for different m values: $m = 20$ ($\nabla \cdots$), $m = 25$ ($+ - \cdot$ line), $m = 30$ ($\circ - -$ line), $m = 35$ ($\times \cdots$ line), $m = 40$ ($\triangleright - \cdot$ line), $m = 45$ ($\square - -$ line), $m = 50$ ($\triangle - \cdot$ line). The solid lines give the value for no restart.

the projected system small.

When we increase the number of active processors by a factor 4, as is done in Problem **4** (see Figure 6), we observe that again a reduction in wall clock time can be achieved by using a well-chosen (k_{min}, m) combination. The number of iterations slightly differ from those given in Figure 5, but the pictures with the Jacobi-Davidson times look similar to those in Figure 5. If we should have enlarged N by a factor of 4 and left the block size unchanged, we may expect execution times as in Figure 5.

For Problem **4**, the limit of 80 iterations seems to be very critical. The right-hand plots of Figure 7 demonstrate that the number of iterations does not decrease monotonously when k_{min} increases for a fixed value m as holds for the previous problems. Moreover, it may happen that for some (k_{min}, m) combination, the limit of JD iterations is too strictly, while for both a smaller and larger k_{min} value the desired N_{ev} eigenpairs were easily found. In the left-hand plots only those results are included, which generate 12 eigenvalues within 80 iterations. Apparently, for the standard case with $m = 57$ and $30 \leq k_{min} \leq 45$, even less iterations are required than in case of no restarts. Of course, this will lead to a time which is far better than for the no-restart case. For the harmonic approach the behavior of the number of JD steps is less obvious, but also here the

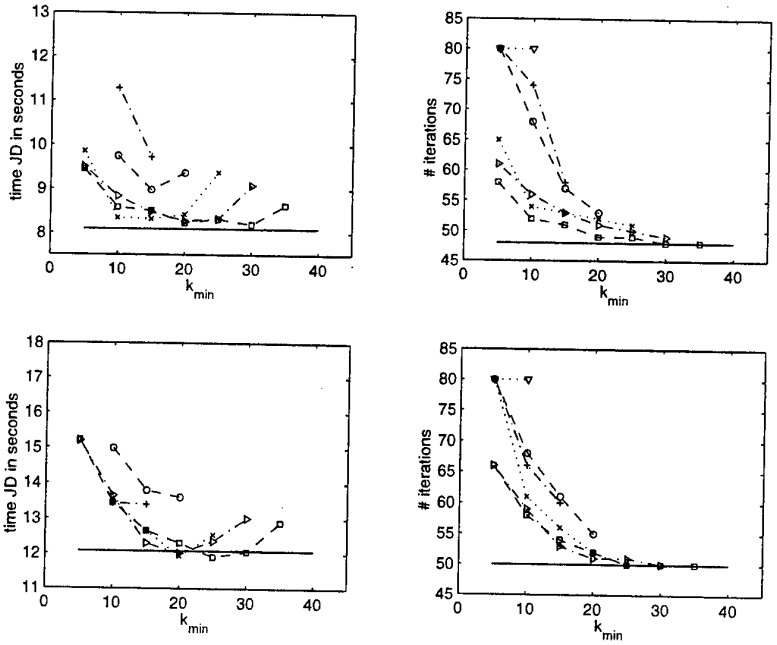


Fig. 5. The upper pictures result on problem **2** using standard Ritz values. The lower pictures result on the same problem with harmonic Ritz values. Results are shown for different m values: $m = 20$ ($\nabla \cdots$), $m = 25$ ($+ - \cdot$ line), $m = 30$ ($\circ - -$ line), $m = 35$ ($\times \cdots$ line), $m = 40$ ($\triangleright - \cdot$ line), $m = 45$ ($\square - -$ line). The solid lines give the value for no restart.

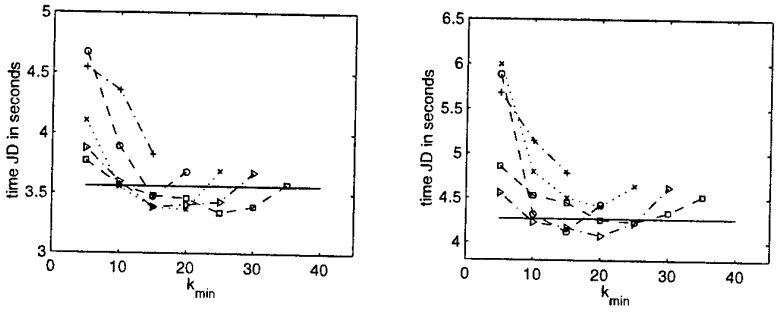


Fig. 6. The left pictures results on problem **3** using standard Ritz values. The right pictures result on the same problem with harmonic Ritz values. Results are shown for different m values: $m = 20$ ($\nabla \cdots$), $m = 25$ ($+ - \cdot$ line), $m = 30$ ($\circ - -$ line), $m = 35$ ($\times \cdots$ line), $m = 40$ ($\triangleright - \cdot$ line), $m = 45$ ($\square - -$ line). The solid lines give the value for no restart.

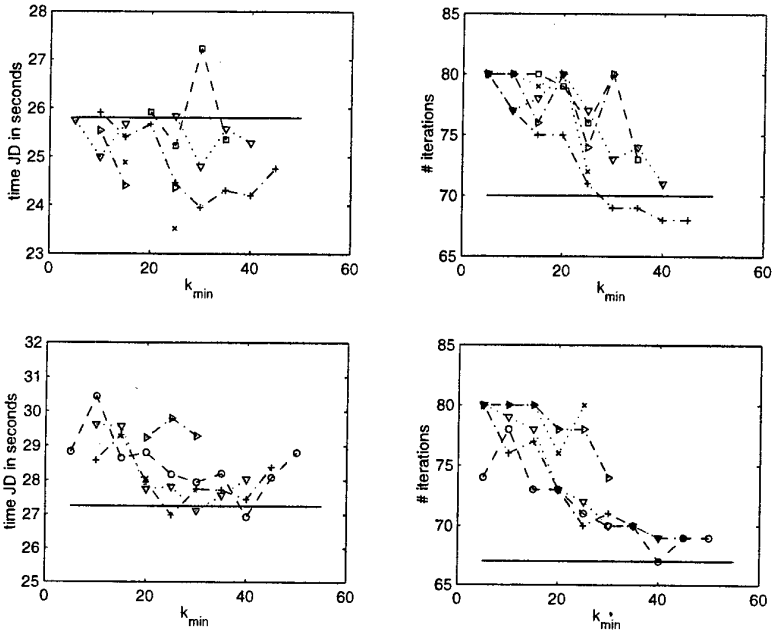


Fig. 7. The upper pictures result on problem **4** using standard Ritz values. The lower pictures result on the same Problem with harmonic Ritz values. Results are shown for different m values: $m = 37$ ($\times \cdots$ line), $m = 42$ ($\triangleright - \cdot$ line), $m = 47$ ($\square - -$ line), $m = 52$ ($\triangledown - \cdot$ line), $m = 57$ ($+ - \cdot$ line), $m = 62$ ($\circ - -$ line). The solid lines give the value for no restart.

monotonicity is lost. Execution times become unpredictable and the conclusion must be that it is better not to restart.

3.5 Parallel execution timing results

Table 1 shows the execution times of several parts of the Jacobi-Davidson algorithm on the Cray T3E; the numbers in parentheses show the Gflop-rates. We took

$$N_{ev} = 20;\ tol_{sJD} = 10^{-8};\ tol_{hJD} = 10^{-5};\ k_{min} = 10;\ m = 30 + N_{ev};\ it_{SOL} = 0.$$

The number of eigenvalues found slightly depends on the number of processors involved: about 11 for the standard and 13 for the harmonic approach within 80 iterations.

The construction of L and U is a very time-consuming part of the algorithm. However, with a well-chosen target σ ten up to twenty eigenvalues can be found within 80 iterations. Hence, the life-time of a (L, U) pair is about 80 iterations. On account of the cyclic reduction part of the LU factorization, a process that

Table 1. Wall clock times in seconds for the standard and harmonic Ritz approach. $N = 4096$, $n = 64$.

p	Preprocessing	Time *standard* JD	Time *harmonic* JD	Triangular solves
32	7.90 (6.75)	64.59	88.61	25.56 (2.08)
64	4.08 (13.21)	31.70	43.78	13.28 (4.02)
128	2.19 (24.78)	15.07	21.33	7.28 (7.36)
256	1.27 (42.69)	8.55	11.48	4.36 (12.29)
512	0.84 (64.65)	5.64	7.02	3.01 (17.81)

starts on all processors, while at each step half of the active processors becomes idle, we may not expect linear speed-up. The fact that the parallel performance of DDCR is quite good is caused by the domain decomposition part of the LU. For more details we refer to [2, 5].

About 40% of the execution time is spent by the computation of the LU factorization (in Table 1 'Preprocessing'), which does not depend on the number of processors. The storage demands for Problem **5** are so large that at least the memories of 32 processors are necessary. DDCR is an order $\mathcal{O}(Nn^3)$ process performed by Level 3 BLAS and it needs less communication: only sub- and super diagonal blocks of size n-by-n must be transfered. As a consequence, for the construction of L and U, the communication time can be neglected also due to the fast communication between processors on the Cray T3E. The Gflop-rates attained for the construction of the LU are impressively high just like its parallel speed-up.

The application of L and U, consisting of two triangular solves, is the most expensive component of the JD process after preprocessing. It parallelizes well, but its speed is much lower, because it is built up of Level 2 BLAS operations. The wall clock times for *standard* and *harmonic* JD are given including the time spent on the triangular solves. Obviously, a harmonic iteration step is more expensive than a standard step, but the overhead becomes less when more processors are used, because the extra operations parallelize very well.

4 Conclusions

We have examined the convergence behaviour of two Jacobi-Davidson variants, one using standard Ritz values, the other one harmonic Ritz values. For the kind of eigenvalue problems we are interested in, arising from MagnetoHydro-Dynamics, both methods converge very fast and parallelize pretty well. With $tol_{sJD} = 10^{-8}$ and $tol_{hJD} = 10^{-5}$ in the acceptance criteria (10) and (11), respectively, both variants give about the same amount of eigenpairs. The harmonic variant is about 20% more expensive, but results into more accurate eigenpairs. With a well-chosen target ten up to twenty eigenvalues can be found. Even

for very large problems, $N_t = 65,536$ and $N_t = 262,144$, we obtain more than 10 sufficient accurate eigenpairs in a few seconds.

Special attention has been paid to a restarting technique. The (k_{min}, m) parameter combination prescribes the amount of information that remains in the system after a restart and the maximum size of the projected system. In this paper we have demonstrated that k_{min} may not be too small. because then too much information gets lost. On the other hand, too large k_{min} values lead to many restarts and become expensive in execution time. In general, the number of iterations decreases when m increases. It depends on the N_t/p value, as we have shown, whether restarts lead to a reduction in the wall clock time for the Jacobi-Davidson process.

Acknowledgments

The authors wish to thank Herman te Riele for many stimulating discussions and suggestions for improving the presentation of the paper. They gratefully acknowledge HPαC (Delft, The Netherlands) for their technical support, and Cray Research for a sponsored account on the Cray T3E (Eagan, MN, USA), and the Dutch National Computing Facilities Foundation NCF for the provision of computer time on the Cray C90 and the Cray T3E.

References

1. W. Kerner, S. Poedts, J.P. Goedbloed, G.T.A. Huysmans, B. Keegan, and E. Schwartz. -. In P. Bachman and D.C. Robinson, editors, *Proceedings of 18th Conference on Controlled Fusion and Plasma Physics.* EPS: Berlin, 1991. IV.89-IV.92.
2. Margreet Nool and Auke van der Ploeg. A Parallel Jacobi-Davidson Method for solving Generalized Eigenvalue Problems in linear Magnetohydrodynamics. Technical Report NM-R9733, CWI, Amsterdam, December 1997.
3. G.L.G. Sleijpen, J.G.L. Booten, D.R. Fokkema, and H.A. van der Vorst. Jacobi-Davidson Type Methods for Generalized Eigenproblems and Polynomial Eigenproblems. *BIT*, 36:595–633, 1996.
4. G.L.G Sleijpen and H.A. van der Vorst. A Jacobi-Davidson iteration method for linear eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 17(2):401–425, april 1996.
5. A. van der Ploeg. Reordering Strategies and LU-decomposition of Block Tridiagonal Matrices for Parallel Processing. Technical Report NM-R9618, CWI, Amsterdam, October 1996.

This article was processed using the LaTeX macro package with LLNCS style

A Level 3 Algorithm for the Symmetric Eigenproblem

Dieter F. Kvasnicka[1], Wilfried N. Gansterer[2], and Christoph W. Ueberhuber[3]

[1] Institute for Technical Electrochemistry, University of Technology, Vienna
dieter@titania.tuwien.ac.at
[2] Institute for Applied and Numerical Mathematics, University of Technology, Vienna
ganst@aurora.tuwien.ac.at
[3] Institute for Applied and Numerical Mathematics, University of Technology, Vienna
christof@uranus.tuwien.ac.at

Abstract. This paper shows how the symmetric eigenproblem, which is the computationally most demanding part of numerous scientific and industrial applications, can be solved much more efficiently than by using algorithms currently implemented in LAPACK routines.
The main techniques used in the algorithm presented in this paper are (i) sophisticated blocking in the tridiagonalization, which leads to a two-sweep algorithm; and (ii) the computation of the eigenvectors of a band matrix instead of a tridiagonal matrix.
This new algorithm improves the locality of data references and leads to a significant improvement in the floating-point performance of symmetric eigensolvers on modern computer systems. Speedup factors of up to four (depending on the computer architecture and the matrix size) have been observed.

Keywords: Numerical Linear Algebra, Symmetric Eigenproblem, Tridiagonalization, Performance Oriented Numerical Algorithm, Blocked Algorithm

1 Introduction

Reducing a dense symmetric matrix A to tridiagonal form T is an important preprocessing step in the solution of the symmetric eigenproblem. LAPACK (Anderson et al. [1]) provides a blocked tridiagonalization routine whose memory reference patterns are not optimal on modern computer architectures. In this LAPACK routine a significant part of the computation is performed by calls to Level 2 BLAS routines. Unfortunately, Level 2 BLAS do not have a ratio of floating-point operations to data movement that is high enough to enable efficient reuse of data that reside in cache or local memory (see Table 1). Thus,

This work was supported by the Austrian Science Foundation (FWF).

software construction based on calls to Level 2 routines is not well suited to computers with a memory hierarchy and multiprocessor machines.

Table 1. Ratio of floating-point operations to data movement for three closely related operations from the Level 1, 2, and 3 BLAS (Dongarra et al. [6])

BLAS	Routine	Memory Accesses	Flops	Flops per Memory Access
Level 1	`daxpy`	$3n$	$2n$	$2/3$
Level 2	`dgemv`	n^2	$2n^2$	2
Level 3	`dgemm`	$4n^2$	$2n^3$	$n/2$

Bischof et al. [4,5] recently developed a general framework for reducing the bandwidth of symmetric matrices, which improves data locality and allows for the use of Level 3 BLAS instead of Level 2 BLAS.

In this paper we introduce an important modification to this framework in case eigenvectors have to be computed, too: Accumulating the transformation information for the eigenvectors incurs an overhead which can outweigh the benefits of a multisweep reduction (as remarked in Bischof et al. [4,5]). Therefore we compute the required eigenvectors directly from the intermediate band matrix. Analyses and experimental results show that our approach to improving memory access patterns is superior to established algorithms in many cases.

2 Eigensolver with Improved Memory Access

A real symmetric $n \times n$ matrix A can be factorized as

$$A = V^T B V = Q^T T Q = Z^T \Lambda Z \tag{1}$$

where V, Q, and Z are orthogonal matrices. B is a symmetric band matrix with band width $2b + 1$, T is a symmetric tridiagonal matrix, and Λ is a diagonal matrix whose diagonal elements are the eigenvalues of the (similar) matrices A, B and T. The column vectors of Z are the eigenvectors of A.

LAPACK reduces a given matrix A to tridiagonal form T by applying Householder similarity transformations. A bisection algorithm[1] is used to compute selected eigenvalues of T, which are also eigenvalues of A. The eigenvectors of T are found by inverse iteration on T. These eigenvectors have to be transformed into the eigenvectors of A using the transformation matrix Q.

The new method proposed does not compute the tridiagonal matrix T from A directly, but derives a band matrix B as an intermediate result. This band reduction can be organized with good data locality, which is critical for high performance on modern computer architectures. Using a block size b in the first

[1] If *all* eigenvalues are required. LAPACK also provides other algorithms.

reduction sweep results in a banded matrix B with a semibandwidth of at least b. This relationship leads to a tradeoff:

- If smaller values of b are chosen, then a larger number of elements of A are eliminated. This decrease in the number of non-zero elements leads to a smaller amount of data to be processed in later steps.
- If larger values of b are chosen, then better performance improvements are obtained in the first reduction sweep.

Using appropriate values of b the two-sweep tridiagonal reduction achieves speedups of up to ten as compared with the LAPACK tridiagonal reduction (see Gansterer, Kvasnicka [7, 8]).

The eigenvectors can be computed by inverse iteration either from A, B, or T. In numerical experiments inverse iteration on B turned out to be the most effective. The eigenvectors of A have to be computed from the eigenvectors of B using the orthogonal transformation matrix V.

The new algorithm includes two special cases:

1. If $V = Q$ then $B = T$, and the inverse iteration is performed on the tridiagonal matrix T. This variant coincides with the LAPACK algorithm.
2. If $V = I$ then $B = A$, and the inverse iteration is performed on the original matrix A. This variant is to be preferred if only a few eigenvectors have to be computed and the corresponding eigenvalues are known.

3 The New Level 3 Eigensolver

The blocked LAPACK approach puts the main emphasis on accumulating several elimination steps. b rank-2 updates (each of them a Level 2 BLAS operation) are aggregated to form one rank-$2b$ update and hence one Level 3 BLAS operation. However, this approach does not take into account memory access patterns in the updating process. We tried to introduce blocking in a much stricter sense, namely by using *blocked access patterns*.

Partitioning matrix A into block columns, and further partitioning each block column into quadratic submatrices makes it possible to process the block-columns and their submatrices the same way that the single columns and their elements are processed in the original unblocked method: all *blocks*[2] below the subdiagonal *block* are eliminated, which leaves a *block* tridiagonal matrix, i.e., a *band* matrix (when considered elementwise). At first sight this matrix has bandwidth $4b - 1$. Further examination shows that the first elimination sweep can be organized such that the subdiagonal blocks in the block tridiagonal matrix are of upper triangular form. Hence the band width of B can be reduced to $2b + 1$.

There are two characteristics which distinguish the newly developed Level 3 algorithm from standard algorithms (see Fig. 1):

[2] Replace *block* by *element* (or *elementwise*) to get the original, unblocked, algorithm.

Two-sweep tridiagonalization: The first sweep reduces the matrix A to a band matrix B. The second sweep reduces B to a tridiagonal matrix T. No fill-in occurs in the first reduction sweep. In the second sweep, however, additional operations are needed to remove fill-in.

Inverse iteration on the band matrix: Calculating the eigenvectors of B avoids the large overhead entailed by the backtransformation of the eigenvectors of T. On the other hand, the overhead caused by inverse iteration on B does not outweigh the benefits of this approach.

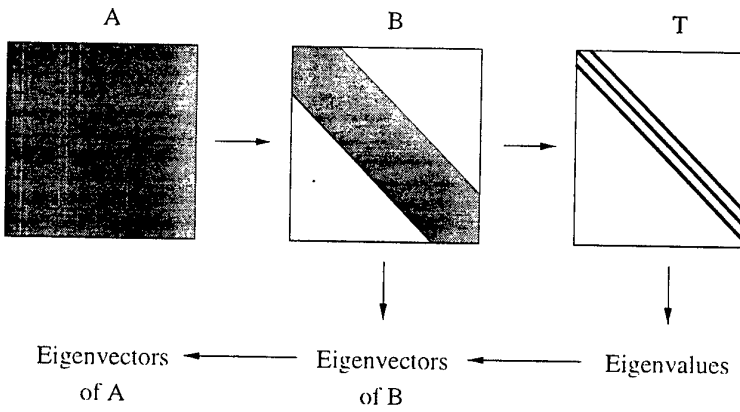


Fig. 1. Basic concept of the new eigensolver

4 Implementation Details

The resulting algorithm has five steps.

1. Reduce the matrix A to the band matrix B.
   (a) Compute the transformation

$$V_i^T A V_i = (I - u_b u_b^T) \cdots (I - u_1 u_1^T) A (I - u_1 u_1^T) \cdots (I - u_b u_b^T)$$

   for a properly chosen subblock of A. The Householder vectors for the transformation of the columns of this subblock do not depend on each other. This independence enables a new way of blocking in Step 1c.

   (b) Collect the Householder vectors as column vectors in an $n \times b$ matrix Y and compute the $n \times b$ matrix W such that the transformation matrix V_i is represented as $(I - WY^T)$. Matrix Y can be stored in those parts of A which have just been eliminated. Matrix W requires separate storage of order $O(n^2)$; therefore it is overwritten and has to be computed again in the backtransformation step.

(c) Perform a rank-$2b$ update using the update matrix $(I - WY^T)$.
(d) Iterate Steps 1a to 1c over the entire matrix A.

2. Reduce the matrix B to the tridiagonal matrix T. Fill-in increases the total number of operations needed for the tridiagonalization.
3. Compute the desired eigenvalues of the tridiagonal matrix T.
4. Compute the corresponding eigenvectors of B by inverse iteration. The computation of the eigenvectors of B requires a higher amount of operations than the computation of the eigenvectors of T.
5. Transform the eigenvectors of B to the eigenvectors of the input matrix A. The update matrices W have to be computed anew. The transformation matrix V, which occurs in (1), is not computed explicitly.

Variations of the representation of V_i are (see Bischof [3])

- $V_i = (I + WY^T)$. The matrix Y holds a set of Householder vectors whereas the matrix W is computed explicitly. This version is actually used in the current implementation.
- $V_i = (I - GG^T)$ (see Schreiber, Parlett [9]). This version needs higher effort for computing G as well as for coding. However, memory for storing W and the redundant effort to compute W twice is saved.
- $V_i = (I - YUY^T)$ (see Schreiber, Van Loan [10]) with a $b \times b$ triangular matrix U. The storage requirement for U is nearly negligible.

The new algorithm and its variants make it possible to use Level 3 BLAS in all computations involving the original matrix, in contrast to the LAPACK routine **dsyevx**. This routine performs 50 % of the operations needed for the tridiagonalization in Level 2 BLAS.

5 Complexity

The total execution time T of our algorithm consists of five parts:

$T_1 \sim c_1(b)n^3$ for reducing the symmetric matrix A to a band matrix B,
$T_2 \sim c_2(b)n^2$ for reducing the band matrix B to a tridiagonal matrix T,
$T_3 \sim c_3(k)n^2$ for computing k eigenvalues of T,
$T_4 \sim c_4(b)kb^2n^2$ for computing the corresponding k eigenvectors of B, and
$T_5 \sim c_5kn^2$ for transforming the eigenvectors of B into the eigenvectors of A.

The parameters c_1, c_2, and c_4 depend on the semibandwidth b. The parameter c_1 decreases in b, whereas c_2 and c_4 increase in b due to an increasing number of operations to be performed. The parameter c_3 depends on the number of computed eigenvectors k; whereas c_5 is independent of the problem size.

The band reduction step is the only part of the algorithm requiring an $O(n^3)$ effort. Thus, a large semibandwidth b of B seems to be desirable. However, b should be chosen appropriately not only to speed up the band reduction (T_1), but also to make the tridiagonalization (T_2) and the eigenvector computation (T_4) as efficient as possible.

For example, on an SGI Power Challenge the calculation of $k = 200$ eigenvalues and eigenvectors of a symmetric 2000×2000 matrix requires a total execution time $T = 87$ s. T_1 is 55 s, i.e., 63 % of the total time. T_4 is 16 s, i.e., 18 % of T. The other steps of the algorithm require an insignificant part of the execution time.

6 Results

A first implementation of our algorithm uses routines from the symmetric band reduction toolbox (SBR; see Bischof et al. [2, 4, 5]), EISPACK routines (Smith et al. [11]), LAPACK routines (Anderson et al. [1]), and some of our own routines.

In numerical experiments we compare the well established LAPACK routine **dsyevx** with the new algorithm. On an SGI Power Challenge (with an R8000 processor running with 90 MHz), speedup factors of up to 4 were observed (see Table 2 and Fig. 5).

Table 2. Execution times (in seconds) on an SGI Power Challenge. $k = n/10$ of the n eigenvalues and eigenvectors were computed

n	k	b	LAPACK **dsyevx**	New Method	Speedup
500	50	6	1.5 s	2.1 s	0.7
1000	100	6	16.6 s	12.6 s	1.3
1500	150	6	74.4 s	39.2 s	1.9
2000	200	10	239.2 s	89.7 s	2.7
3000	300	12	945.5 s	286.4 s	3.3
4000	400	12	2432.4 s	660.5 s	3.7

Fig. 2 shows the normalized computing time $T(n)/n^3$. The significant speedup of the new algorithm when applied to large problems is striking. This speedup has nothing to do with complexity, which is nearly identical for both algorithms (see Fig. 3). The reason for the good performance of the new algorithm is its significantly improved utilization of the computer's potential peak performance (see Fig. 4). The deteriorating efficiency of the LAPACK routine (due to cache effects) causes the $O(n^4)$ behavior of its computation time between $n = 500$ and $n = 2000$ (see Fig. 2).

The new algorithm shows significant speedups compared to existing algorithms for large matrices ($n \geq 1500$) and a small subset of eigenvalues and eigenvectors ($k = n/10$) on all computers at our disposal, including workstations of DEC, HP, IBM, and SGI.

Choosing the block size b. Experiments show that the optimum block size b, which equals the smallest possible band width, increases slightly with larger

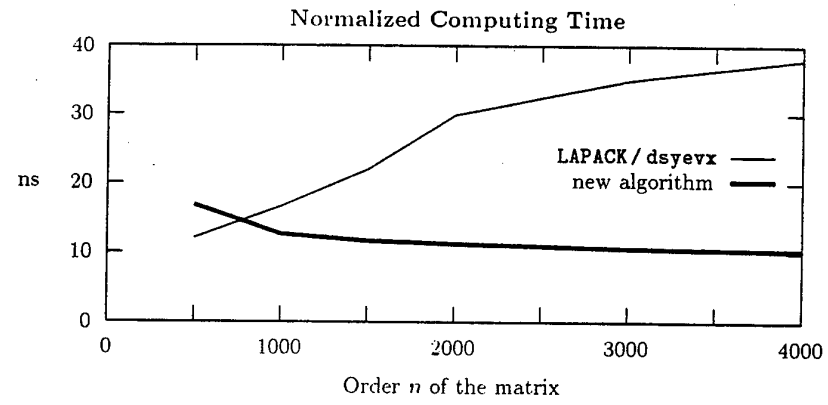


Fig. 2. Normalized computing time $T(n)/n^3$ in nanoseconds

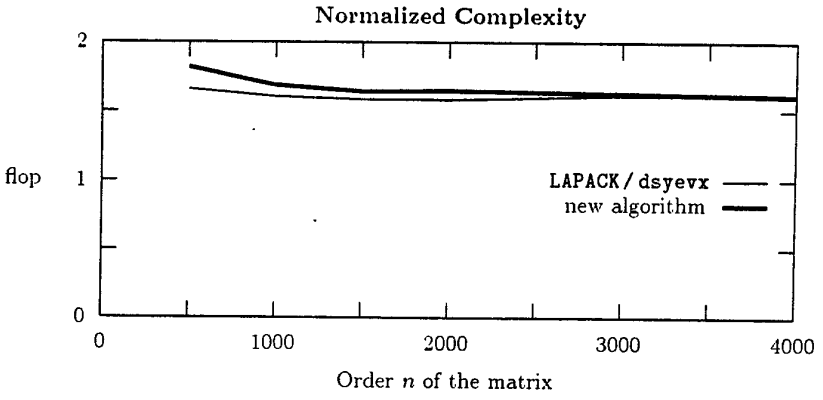


Fig. 3. Normalized number $\mathrm{op}(n)/n^3$ of floating-point operations

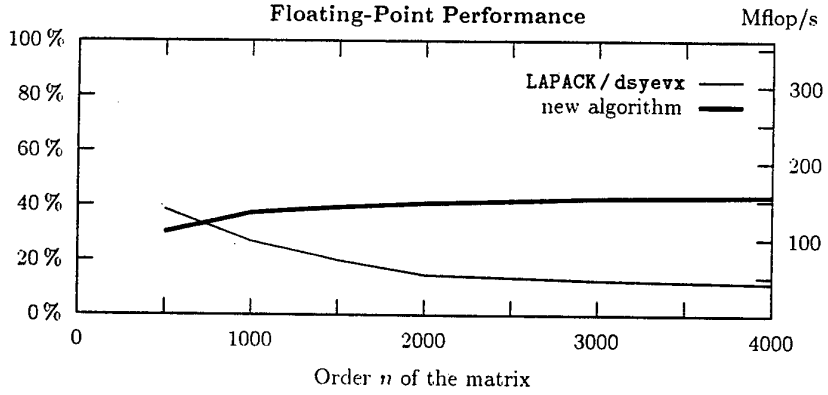


Fig. 4. Floating-point performance (MFlop/s) and efficiency (%)

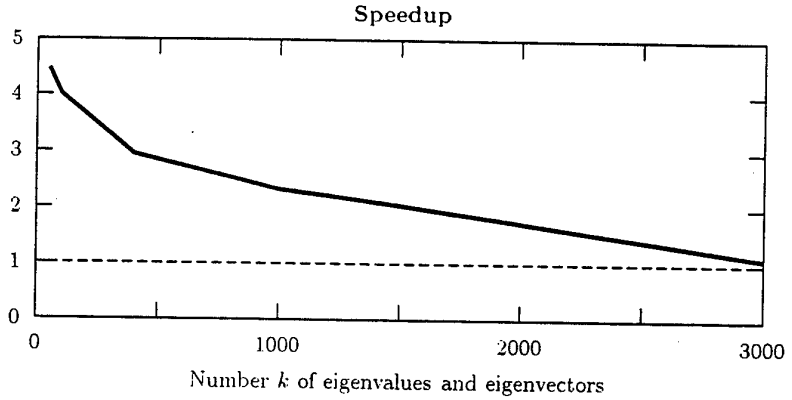


Fig. 5. Speedup of the new algorithm as compared with the LAPACK routine **dsyevx**. Performance improvements are all the better if k is only a small percentage of the $n = 3000$ eigenvalues and eigenvectors

matrix sizes (see Table 2). A matrix of order 500 requires a b of only 4 or 6 for optimum performance, depending on the architecture (and the size of cache lines). Band widths of 12 or 16 are optimum on most architectures when the matrix order is larger than 2000. A hierarchically blocked version which is currently under development allows increasing the block size without increasing the band width and therefore leads to even higher performance.

7 Conclusion

We presented an algorithm for the computation of selected eigenvalues and eigenvectors of symmetric matrices which is significantly faster than existing algorithms. This speedup is achieved by improved blocking in the tridiagonalization process, which significantly improves data locality.

LAPACK algorithms for the symmetric eigenproblem spend up to 80 % of their execution time in Level 2 BLAS, which do not perform well on cache-based and multiprocessor computers. In our algorithm all performance relevant steps make use of Level 3 BLAS.

The price that has to be paid for the improved performance in the tridiagonalization process is that the eigenvectors cannot be computed from the tridiagonal matrix because of prohibitive overhead in the backtransformation. They have to be computed from the intermediate band matrix.

When choosing the block size, a compromise has to be made: Larger block sizes improve the performance of the band reduction. Smaller block sizes have to be used to reduce the band width and, therefore, to speed up the inverse iteration on the band matrix. This property makes the new algorithm particularly attractive (on most architectures) if not all eigenvectors are required.

If the gap between processor speed and memory bandwidth further increases, our algorithm will be highly competitive also for solving problems where all eigenvectors are required.

Future Development. Routines dominated by Level 3 BLAS operations, like the eigensolver presented in this paper, have the potential of speeding up almost linearly on parallel machines. That is why we are currently developing a parallel version. Another promising possibility of development is the use of hierarchical blocking (see Ueberhuber [12]).

References

1. E. Anderson et al., LAPACK *Users' Guide. 2nd ed.*, SIAM Press, Philadelphia, 1995.
2. C. H. Bischof, B. Lang, X. Sun, *Parallel Tridiagonalization through Two-Step Band Reduction*, Proceedings of the Scalable High-Performance Computing Conference, IEEE Press, Washington, D. C., 1994, pp. 23–27.
3. C. H. Bischof, *A Summary of Block Schemes for Reducing a General Matrix to Hessenberg Form*, Technical Report ANL/MCS-TM-175, Argonne National Laboratory, 1993.
4. C. H. Bischof, B. Lang, X. Sun, *A Framework for Symmetric Band Reduction*, ACM Trans. Math. Software, Argonne Preprint ANL/MCS-P586-0496 (1996).
5. C. H. Bischof, B. Lang, X. Sun, *The SBR Toolbox – Software for Successive Band Reduction*, ACM Trans. Math. Software, Argonne Preprint ANL/MCS-P587-0496 (1996).
6. J. J. Dongarra et al., *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM Press, Philadelphia, 1991.
7. W. Gansterer, D. Kvasnicka, *High-Performance Computing in Material Sciences. The Standard Eigenproblem – Concepts*, Technical Report 3/97, AURORA-5, Technical University of Vienna, 1997.
8. W. Gansterer, D. Kvasnicka, *High-Performance Computing in Material Sciences. The Standard Eigenproblem – Experiments*, Technical Report 5/97, AURORA-5, Technical University of Vienna, 1997.
9. R. Schreiber, B. Parlett, *Block Reflectors: Theory and Computation*, SIAM J. Numer. Anal. 25 (1988), pp. 189–205.
10. R. Schreiber, C. Van Loan, *A Storage-Efficient WY Representation for Products of Householder Transformations*, SIAM J. Sci. Stat. Comput. 10–1 (1989), pp. 53–57.
11. B. T. Smith et al., *Matrix Eigensystem Routines* – EISPACK *Guide*, Lecture Notes in Computer Science, Vol. 6, Springer-Verlag, Berlin Heidelberg New York Tokyo, 1976.
12. C. W. Ueberhuber, *Numerical Computation*, Springer-Verlag, Berlin Heidelberg New York Tokyo, 1997.

Synchronous and asynchronous parallel algorithms with overlap for almost linear systems

Josep Arnal, Violeta Migallón, and José Penadés

Departamento de Ciencia de la Computación e Inteligencia Artificial,
Universidad de Alicante,
E-03071 Alicante, Spain
{arnal, violeta, jpenades}@dtic.ua.es

Abstract. Parallel algorithms for solving almost linear systems are studied. A non-stationary parallel algorithm based on the multisplitting technique and its extension to an asynchronous model are considered. Convergence properties of these methods are studied for M-matrices and H-matrices. We implemented these algorithms on two distributed memory multiprocessors, where we studied their performance in relation to overlapping of the splittings at each iteration.

1 Introduction

We are interested in the parallel solution of almost linear systems of the form

$$Ax + \Phi(x) = b, \tag{1}$$

where $A = (a_{ij})$ is a real $n \times n$ matrix, x and b are n-vectors and $\Phi : \mathbb{R}^n \to \mathbb{R}^n$ is a nonlinear diagonal mapping (i.e., the ith component Φ_i of Φ is a function only of x_i).

These systems appear in practice from the discretization of differential equations, which arise in many fields of applications such as trajectory calculation or the study of oscillatory systems; see e.g., [3], [5] for some examples.

Considering that system (1) has in fact a unique solution, White [18] introduced the parallel nonlinear Gauss-Seidel algorithm, based on both the classical nonlinear Gauss-Seidel method (see [13]) and the multisplitting technique (see [12]). Until then, the multisplitting technique had only been used for linear problems. Recently, in the context of relaxed methods, Bai [1] has presented a class of algorithms, called parallel nonlinear AOR methods, for solving system (1). These methods are a generalization of the parallel nonlinear Gauss-Seidel algorithm [18].

In order to get a good performance of all processors and a good load balance among processors, in this paper we extend the idea of the non-stationary methods to the problem of solving the almost linear system (1). This technique was introduced in [6] for solving linear systems, (see also [8], [11]). In a formal way,

let us consider a collection of splittings $A = (D - L_{\ell,k,m}) - U_{\ell,k,m}$, $\ell = 1, 2, \ldots$, $k = 1, 2, \ldots, \alpha$, $m = 1, 2, \ldots, q(\ell, k)$, such that $D = \mathrm{diag}(A)$ is nonsingular and $L_{\ell,k,m}$ are strictly lower triangular matrices. Note that matrices $U_{\ell,k,m}$ are not generally upper triangular. Let E_k be nonnegative diagonal matrices such that $\sum_{k=1}^{\alpha} E_k = I$.

Let us define $r_i : \mathbb{R} \to \mathbb{R}$, $1 \leq i \leq n$, such that

$$r_i(t) = a_{ii}t + \Phi_i(t), \quad t \in \mathbb{R}, \tag{2}$$

and suppose that there exists the inverse function of each r_i, denoted by r_i^{-1}.

Let us consider the operators $P_{\ell,k,m} : \mathbb{R}^n \to \mathbb{R}^n$ such that each of them maps x into y in the following way

$$\begin{cases} y_i = \omega \hat{y}_i + (1-\omega)x_i, \ 1 \leq i \leq n, \ \omega \in \mathbb{R}, \ \omega \neq 0, \\ \text{and } \hat{y}_i = r_i^{-1}(z_i), \text{ with} \\ z = \mu L_{\ell,k,m}\hat{y} + (1-\mu)L_{\ell,k,m}x + U_{\ell,k,m}x + b, \ \mu \in \mathbb{R}. \end{cases} \tag{3}$$

With this notation, the following algorithm describes a non-stationary parallel nonlinear method to solve system (1). This algorithm is based on the AOR-type methods. It is assumed that processors update their local approximation as many times as the *non-stationary* parameters $q(\ell, k)$ indicate.

Algorithm 1 (NON-STATIONARY PARALLEL NONLINEAR ALG.).
Given the initial vector $x^{(0)}$, and a sequence of numbers of local iterations $q(\ell, k)$, $\ell = 1, 2, \ldots$, $k = 1, 2, \ldots, \alpha$

For $\ell = 1, 2, \ldots$, until convergence
  In processor k, $k = 1$ to α
    $x^{\ell,k,0} = x^{(\ell-1)}$
    For $m = 1$ to $q(\ell, k)$
      $x^{\ell,k,m} = P_{\ell,k,m}(x^{\ell,k,m-1})$
  $x^{(\ell)} = \sum_{k=1}^{\alpha} E_k x^{\ell,k,q(\ell,k)}$.

We note that Algorithm 1 extends the nonlinear algorithms introduced in [1] and [18]. Moreover, Algorithm 1 reduces to Algorithm 2 in [8], when $\Phi(x) = 0$ and for all $\ell = 1, 2, \ldots$, $m = 1, 2, \ldots, q(\ell, k)$, $L_{\ell,k,m} = L_k$ and $U_{\ell,k,m} = U_k$, $k = 1, 2, \ldots, \alpha$. Here, the formulation of Algorithm 1 allows us to use different splittings not only in each processor but at each global iteration ℓ and/or at each local iteration m. Furthermore, the overlap is allowed as well.

In this algorithm all processors complete their local iterations before updating the global approximation $x^{(\ell)}$. Thus, this algorithm is synchronous.

To construct an asynchronous version of Algorithm 1 we consider an iterative scheme on $\mathbb{R}^{\alpha n}$. More precisely, we consider that, at the ℓth iteration, processor k performs the calculations corresponding to its $q(\ell, k)$ splittings, saving the

update vector in $x_k^{(\ell)}$, $k = 1, 2, \ldots, \alpha$. Moreover, at each step, processors make use of the most recent vectors computed by the other processors, which are previously weighted with the matrices E_k, $k = 1, 2, \ldots, \alpha$.

In a formal way, let us define the sets $J_\ell \subseteq \{1, 2, \ldots, \alpha\}$, $\ell = 1, 2, \ldots$, as $k \in J_\ell$ if the kth part of the iteration vector is computed at the ℓth step. The superscripts $r(\ell, k)$ denote the iteration number in which the processor k computed the vector used at the beginning of the ℓth iteration.

As it is customary in the description and analysis of asynchronous algorithms (see e.g., [2], [4]), we always assume that the superscripts $r(\ell, k)$ and the sets J_ℓ satisfy the following conditions

$$r(\ell, k) < \ell \text{ for all } k = 1, 2, \ldots, \alpha, \ \ell = 1, 2, \ldots. \tag{4}$$

$$\lim_{\ell \to \infty} r(\ell, k) = \infty \text{ for all } k = 1, 2, \ldots, \alpha. \tag{5}$$

$$\text{The set } \{\ell \mid k \in J_\ell\} \text{ is unbounded for all } k = 1, 2, \ldots, \alpha. \tag{6}$$

Let us consider the operators $P_{\ell,k,m}$ used in Algorithm 1. With this notation, the asynchronous counterpart of that algorithm corresponds to the following algorithm.

Algorithm 2 (ASYNC. NON-STATIONARY PARALLEL NONLINEAR ALG.). Given the initial vectors $x_k^{(0)}$, $k = 1, 2, \ldots, \alpha$, and a sequence of numbers of local iterations $q(\ell, k)$, $\ell = 1, 2, \ldots$, $k = 1, 2, \ldots, \alpha$

For $\ell = 1, 2, \ldots$, until convergence

$$x_k^{(\ell)} = \begin{cases} x_k^{(\ell-1)} & \text{if } k \notin J_\ell \\ P_{\ell,k,q(\ell,k)} \cdot \ldots \cdot P_{\ell,k,2} \cdot P_{\ell,k,1} \left(\displaystyle\sum_{j=1}^{\alpha} E_j x_j^{(r(\ell,j))} \right) & \text{if } k \in J_\ell. \end{cases} \tag{7}$$

Note that Algorithm 2 computes iterate vectors of size αn, while it only uses n-vectors to perform the updates. For that reason, from the experimental point of view, we can consider that the sequence of iterate vectors is made up of that n-vectors, that is, $\displaystyle\sum_{j=1}^{\alpha} E_j x_j^{(r(\ell,j))}$, $\ell = 1, 2, \ldots$. Another consequence of what has been mentioned above is that only components of the vectors $x_k^{(\ell)}$ corresponding to nonzero diagonal entries of the matrix E_k need to be computed. Then, the local storage is of order n and not αn.

In order to rewrite the asynchronous iteration (7) more clearly, we define the operators $G^{(\ell)} = (G_1^{(\ell)}, \ldots, G_\alpha^{(\ell)})$, with $G_k^{(\ell)} : \mathbb{R}^{\alpha n} \to \mathbb{R}^n$ such that, if $\tilde{y} \in \mathbb{R}^{\alpha n}$

$$G_k^{(\ell)}(\tilde{y}) = P_{\ell,k,q(\ell,k)} \cdot \ldots \cdot P_{\ell,k,2} \cdot P_{\ell,k,1}(Q\tilde{y}), \quad k = 1, 2, \ldots, \alpha,$$

where

$$Q = [E_1, \ldots, E_k, \ldots, E_\alpha] \in \mathbb{R}^{n \times \alpha n}. \tag{8}$$

Then, iteration (7) can be rewritten as the following iteration

$$x_k^{(\ell)} = \begin{cases} x_k^{(\ell-1)} & \text{if } k \notin J_\ell \\ G_k^{(\ell)}\left(x_1^{(r(\ell,1))}, \ldots, x_k^{(r(\ell,k))}, \ldots, x_\alpha^{(r(\ell,\alpha))}\right) & \text{if } k \in J_\ell. \end{cases} \tag{9}$$

In Section 2, we study the convergence properties of the above algorithms when the matrix in question is either M-matrix or H-matrix. The last section contains computational results which illustrate the behavior of these algorithms on two distributed multiprocessors. In the rest of this section we introduce some notation, definitions and preliminary results.

We say that a vector $x \in \mathbb{R}^n$ is nonnegative (positive), denoted $x \geq 0$ ($x > 0$), if all its entries are nonnegative (positive). Similarly, if $x, y \in \mathbb{R}^n$, $x \geq y$ ($x > y$) means that $x - y \geq 0$ ($x - y > 0$). Given a vector $x \in \mathbb{R}^n$, $|x|$ denotes the vector whose components are the absolute values of the corresponding components of x. These definitions carry over immediately to matrices.

A nonsingular matrix A is said to be an M-matrix if it has non-positive off-diagonal entries and it is monotone, i.e., $A^{-1} \geq O$; see e.g., Berman and Plemmons [3] or Varga [17]. Given a matrix $A = (a_{ij}) \in \mathbb{R}^{n\times n}$, its comparison matrix is defined by $\langle A \rangle = (\alpha_{ij})$, $\alpha_{ii} = |a_{ii}|$, $\alpha_{ij} = -|a_{ij}|$, $i \neq j$. A is said to be an H-matrix if $\langle A \rangle$ is a nonsingular M-matrix.

Lemma 1. *Let $H^{(1)}, H^{(2)}, \ldots, H^{(\ell)}, \ldots$ be a sequence of nonnegative matrices in $\mathbb{R}^{n\times n}$. If there exists a real number $0 \leq \theta < 1$, and a vector $v > 0$ in $\mathbb{R}^n$, such that*

$$H^{(\ell)} v \leq \theta v, \ \ell = 1, 2, \ldots,$$

then $\rho(K_\ell) \leq \theta^\ell < 1$, where $K_\ell = H^{(\ell)} H^{(\ell-1)} \cdots H^{(1)}$, and therefore $\lim_{\ell\to\infty} K_\ell = O$.

Proof. The proof of this lemma can be found, e.g., in [15].

Lemma 2. *Let $A = (a_{ij}) \in \mathbb{R}^{n\times n}$ be an H-matrix and let $\Phi : \mathbb{R}^n \to \mathbb{R}^n$ be a continuous and diagonal mapping. If $sign(a_{ii})\,(t - s)\,(\Phi_i(t) - \Phi_i(s)) \geq 0$, $i = 1, 2, \ldots, n$, for all $t, s \in \mathbb{R}$, then the almost linear system (1) has a unique solution.*

Proof. It is essentially the proof of [1, Lemma 2].

2 Convergence

In order to analyze the convergence of Algorithm 1 we rewrite it as the following iteration scheme

$$x^{(\ell)} = \sum_{k=1}^{\alpha} E_k x^{\ell,k,q(\ell,k)}, \ \ \ell = 1, 2, \ldots, \tag{10}$$

where $x^{\ell,k,q(\ell,k)}$ is computed according to the iteration

$$x^{\ell,k,0} = x^{(\ell-1)}$$

For $m = 1$ to $q(\ell,k)$

$$x_i^{\ell,k,m} = \omega \hat{x}_i^{\ell,k,m} + (1-\omega) x_i^{\ell,k,m-1},\ 1 \le i \le n,\ \omega \in \mathbb{R},\ \omega \neq 0 \qquad (11)$$

and $\hat{x}_i^{\ell,k,m}$ is determined by

$$\hat{x}_i^{\ell,k,m} = r_i^{-1}(z_i^{\ell,k,m}), \qquad (12)$$

where r_i is defined in (2) and

$$z^{\ell,k,m} = \mu L_{\ell,k,m} \hat{x}^{\ell,k,m} + (1-\mu) L_{\ell,k,m} x^{\ell,k,m-1} + U_{\ell,k,m} x^{\ell,k,m-1} + b,\ \mu \in \mathbb{R}.$$

The following theorem ensures the existence of a unique solution of system (1) and shows the convergence of scheme (10) (or Algorithm 1) when A is an H-matrix and $0 \le \mu \le \omega < \frac{2}{1+\rho(|D|^{-1}|B|)}$, with $\omega \neq 0$, where $D = \mathrm{diag}(A)$ and $A = D - B$. Note that, from [17, Theorem 3.10], $|D|$ is a nonsingular matrix and $\rho(|D|^{-1}|B|) < 1$.

Theorem 1. *Let $A = D - L_{\ell,k,m} - U_{\ell,k,m} = D - B$, $\ell = 1,2,\ldots$, $k = 1,2,\ldots,\alpha$, $m = 1,2,\ldots,q(\ell,k)$, be an H-matrix, where $D = \mathrm{diag}(A)$ and $L_{\ell,k,m}$ are strictly lower triangular matrices. Assume that $|B| = |L_{\ell,k,m}| + |U_{\ell,k,m}|$. Let Φ be a continuous and diagonal mapping satisfying*

$$sign(a_{ii})\,(t-s)\,(\Phi_i(t) - \Phi_i(s)) \ge 0,\quad i = 1,2,\ldots,n,\quad \textit{for all } t,s \in \mathbb{R}. \qquad (13)$$

If $0 \le \mu \le \omega < \frac{2}{1+\rho}$, with $\omega \neq 0$, where $\rho = \rho(|D|^{-1}|B|)$, and $q(\ell,k) \ge 1$, $\ell = 1,2,\ldots$, $k = 1,2,\ldots,\alpha$, then the iteration (10) is well-defined and converges to the unique solution of the almost linear system (1), for every initial vector $x^{(0)} \in \mathbb{R}^n$.

Proof. Since Φ_i, $1 \le i \le n$, are continuous mappings satisfying (13), it follows that each r_i given in (2) is one-to-one and maps $\mathbb{R}$ onto $\mathbb{R}$. Hence, each r_i has an inverse function defined in all of $\mathbb{R}$ and thus iteration (10) is well-defined. On the other hand, by Lemma 2, system (1) has a unique solution, denoted x^*. Let $\varepsilon^{(\ell)} = x^{(\ell)} - x^*$ be the error vector at the ℓth iteration of scheme (10). Then, $|\varepsilon^{(\ell)}| \le \sum_{k=1}^{\alpha} E_k \left| x^{\ell,k,q(\ell,k)} - x^* \right|$. Using (13) and reasoning in a similar way as in the proof of [13, Theorem 13.13], it is easy to prove that $|a_{ii}|\,|y - \bar{y}| \le |r_i(y) - r_i(\bar{y})|$, for all $y, \bar{y} \in \mathbb{R}$, where r_i is defined in (2). Therefore, we obtain that $|a_{ii}|\left|r_i^{-1}(z) - r_i^{-1}(\bar{z})\right| \le |z - \bar{z}|$, for all $z, \bar{z} \in \mathbb{R}$. Then, from (12) and using the fact that $x_i^* = r_i^{-1}([\mu L_{\ell,k,m} x^* + (1-\mu)\, L_{\ell,k,m} x^* + U_{\ell,k,m} x^* + b]_i)$, we obtain, for each $i = 1,2,\ldots,n$

$$\begin{aligned} |a_{ii}| \left| \hat{x}_i^{\ell,k,m} - x_i^* \right| &= |a_{ii}| \left| r_i^{-1}(z_i^{\ell,k,m}) - r_i^{-1}(z_i^k) \right| \le \left| z_i^{\ell,k,m} - z_i^k \right| \\ &= \left| \left[\mu L_{\ell,k,m} \left(\hat{x}^{\ell,k,m} - x^* \right) + (1-\mu)\, L_{\ell,k,m} \left(x^{\ell,k,m-1} - x^* \right) \right.\right. \\ &\quad \left.\left. + U_{\ell,k,m} \left(x^{\ell,k,m-1} - x^* \right) \right]_i \right|. \end{aligned}$$

Since these inequalities are true for all $i = 1, 2, \ldots, n$, we can write

$$|D|\left|x^{\ell,k,m} - x^*\right| \leq \left|\mu L_{\ell,k,m}\left(\hat{x}^{\ell,k,m} - x^*\right) + (1-\mu)\, L_{\ell,k,m}\left(x^{\ell,k,m-1} - x^*\right) + U_{\ell,k,m}\left(x^{\ell,k,m-1} - x^*\right)\right|.$$

Since $(|D| - \mu|L_{\ell,k,m}|)^{-1} \geq O$, making use of (11) we obtain, after some algebraic manipulations, that

$$\left|x^{\ell,k,m} - x^*\right| \leq (|D| - \mu\,|L_{\ell,k,m}|)^{-1}\left((\omega-\mu)\,|L_{\ell,k,m}| + \omega\,|U_{\ell,k,m}| + |1-\omega|\,|D|\right)\left|x^{\ell,k,m-1} - x^*\right|, \quad m = 1, 2, \ldots, q(\ell,k).$$

Therefore, $\left|x^{\ell,k,q(\ell,k)} - x^*\right| \leq H_k^{(\ell)}\left|x^{(\ell-1)} - x^*\right|$, where $H_k^{(\ell)} = H_{\ell,k,q(\ell,k)} \cdot \ldots \cdot H_{\ell,k,2} \cdot H_{\ell,k,1}$, and

$$H_{\ell,k,m} = (|D| - \mu\,|L_{\ell,k,m}|)^{-1}\left((\omega-\mu)\,|L_{\ell,k,m}| + \omega\,|U_{\ell,k,m}| + |1-\omega|\,|D|\right). \quad (14)$$

Then $\left|\varepsilon^{(\ell)}\right| \leq H^{(\ell)}|\varepsilon^{(\ell-1)}| \leq \cdots \leq H^{(\ell)} \ldots H^{(1)}|\varepsilon^{(0)}|$, where $H^{(\ell)} = \sum_{k=1}^{\alpha} E_k H_k^{(\ell)}$. Since A is an H-matrix, following the proof of [8, Theorem 4.1] we conclude that for $0 \leq \mu \leq \omega < \frac{2}{1+\rho}$, with $\omega \neq 0$, there exist real constants $0 \leq \theta_k < 1$ and a positive vector v such that $H_k^{(\ell)} v \leq \theta_k v$. Hence, setting $\theta = \max_{k=1,2,\ldots,\alpha} \theta_k$, it obtains $H^{(\ell)} v \leq \theta v$. Then, from Lemma 1 the product $H^{(\ell)} H^{(\ell-1)} \ldots H^{(1)}$ tends to the null matrix as $\ell \to \infty$ and thus $\lim_{\ell\to\infty} \varepsilon^{(\ell)} = 0$. Therefore, the proof is done.

Next we show the convergence of the asynchronous Algorithm 2 under similar hypotheses as in the synchronous case.

Theorem 2. *Let $A = D - L_{\ell,k,m} - U_{\ell,k,m} = D - B$, $\ell = 1, 2, \ldots$, $k = 1, 2, \ldots, \alpha$, $m = 1, 2, \ldots, q(\ell, k)$, be an H-matrix, where $D = \mathrm{diag}(A)$ and $L_{\ell,k,m}$ are strictly lower triangular matrices. Assume that $|B| = |L_{\ell,k,m}| + |U_{\ell,k,m}|$. Let Φ be a continuous and diagonal mapping satisfying for all $t, s \in \mathbb{R}$*

$$sign(a_{ii})\,(t-s)\,(\Phi_i(t) - \Phi_i(s)) \geq 0, \quad i = 1, 2, \ldots, n.$$

Assume further that the sequence $r(\ell, k)$ and the sets J_ℓ, $k = 1, 2, \ldots, \alpha$, $l = 1, 2, \ldots$, satisfy conditions (4–6). If $0 \leq \mu \leq \omega < \frac{2}{1+\rho}$, with $\omega \neq 0$, where $\rho = \rho(|D|^{-1}|B|)$, and $q(\ell, k) \geq 1$, $\ell = 1, 2, \ldots$, $k = 1, 2, \ldots, \alpha$, then the asynchronous Algorithm 2 is well-defined and converges to $\tilde{x}^* = (x^{*T}, \ldots, x^{*T})^T \in \mathbb{R}^{\alpha n}$, where x^* is the unique solution of the almost linear system (1), for all initial vectors $x_k^{(0)} \in \mathbb{R}^n$, $k = 1, 2, \ldots, \alpha$.

Proof. By Lemma 2, the existence and uniqueness of a solution of system (1) is guaranteed. From the proof of Theorem 1 it follows that Algorithm 2 is well-defined. Moreover, there exists a positive vector v and a constant $0 \leq \theta < 1$ such that

$$H_k^{(\ell)} v \leq \theta v, \quad k = 1, 2, \ldots, \alpha, \quad \ell = 1, 2, \ldots. \quad (15)$$

Let us consider $\tilde{v} = (v^T, \ldots, v^T)^T \in \mathbb{R}^{\alpha n}$. As $G_k^{(\ell)}(\tilde{x}^*) = x^*$, then $\tilde{x}^*$ is a fixed point of $G^{(\ell)}$, $\ell = 1, 2, \ldots$. Following the proof of Theorem 1 it easy to prove that

$$|G_k^{(\ell)}(\tilde{y}) - G_k^{(\ell)}(\tilde{z})| \leq H_k^{(\ell)} Q |\tilde{y} - \tilde{z}|, \text{ for all } \tilde{y}, \tilde{z} \in \mathbb{R}^{\alpha n},$$

where Q is defined in (8). Then,

$$|G^{(\ell)}(\tilde{y}) - G^{(\ell)}(\tilde{z})| \leq T^{(\ell)} |\tilde{y} - \tilde{z}|, \text{ for all } \tilde{y}, \tilde{z} \in \mathbb{R}^{\alpha n},$$

where

$$T^{(\ell)} = \begin{bmatrix} H_1^{(\ell)} Q \\ \vdots \\ H_\alpha^{(\ell)} Q \end{bmatrix} \in \mathbb{R}^{\alpha n \times \alpha n}. \tag{16}$$

From equations (15) and (16) it follows that

$$T^{(\ell)} \tilde{v} \leq \theta \tilde{v}, \ \ell = 1, 2, \ldots. \tag{17}$$

Due to the uniformity assumption (17), we can apply [2, Theorem 1] to our case in which the operators change with the iteration superscript. Then, the convergence is shown.

Note that, in the particular case in which A is an M-matrix, condition (13) is reduced to state that the mapping Φ is nondecreasing. Moreover, condition $|B| = |L_{\ell,k,m}| + |U_{\ell,k,m}|$ is equivalent to assume that $L_{\ell,k,m}$ and $U_{\ell,k,m}$ are nonnegative matrices.

3 Numerical experiments

We have implemented the above algorithms on two distributed multiprocessors. The first platform is an IBM RS/6000 SP with 8 nodes. These nodes are 120 MHz Power2 Super Chip (Thin SC) and they are connected through a high performance switch with latency time of 40 microseconds and a bandwidth of 30 to 35 Mbytes per second. The second platform is an ethernet network of five 120 MHz Pentiums. The peak performance of this network is 100 Mbytes per second with a bandwidth around 6.5 Mbytes per second. In order to manage the parallel environment we have used the PVMe library of parallel routines for the IBM RS/6000 SP and the PVM library for the cluster of Pentiums [9], [10].

In order to illustrate the behavior of the above algorithms, we have considered the following semilinear elliptic partial differential equation (see e.g., [7], [16], [18])

$$\begin{aligned} -(K^1 u_x)_x - (K^2 u_y)_y &= -g e^u \quad (x, y) \in \Omega, \\ u &= x^2 + y^2 \ (x, y) \in \partial\Omega, \end{aligned} \tag{18}$$

where

$$\begin{aligned} K^1 &= K^1(x, y) = 1 + x^2 + y^2, \\ K^2 &= K^2(x, y) = 1 + e^x + e^y, \\ g &= g(x, y) = 2(2 + 3x^2 + y^2 + e^x + (1 + y)e^y) e^{-x^2 - y^2}, \\ \Omega &= (0, 1) \times (0, 1). \end{aligned}$$

It is well known that this problem has the unique solution $u(x,y) = x^2 + y^2$. To solve equation (18) using the finite difference method, we consider a grid in Ω of d^2 nodes equally spaced by $h = \Delta x = \Delta y = \frac{1}{d+1}$. This discretization yields an almost linear system $Ax + \Phi(x) = b$, where A is a block tridiagonal symmetric matrix $A = (D_{i-1}, T_i, D_i)_{i=1}^{d}$, where T_i are tridiagonal matrices of size $d \times d$, $i = 1, 2, \ldots, d$, and D_i are $d \times d$ diagonal matrices, $i = 1 \ldots, d-1$; see e.g., [7].

Let $S = \{1, 2, \ldots, n\}$ and let n_k, $k = 1, 2, \ldots, \alpha$, be positive integers which add n. Consider $S_{k,m}$, $k = 1, 2, \ldots, \alpha$, $m = 1, 2, \ldots, q(\ell, k)$, subsets of S defined as

$$S_{k,m} = \{s^1_{k,m}, s^1_{k,m} + 1, \ldots, s^2_{k,m}\}, \tag{19}$$

where

$$\begin{cases} s^1_{k,m} = \max\{1, 1 + \sum_{i<k} n_i - bd - (m-1)d\}, \text{ and} \\ s^2_{k,m} = \min\{n, \sum_{i \le k} n_i + bd + (m-1)d\}, \end{cases} \tag{20}$$

with b being a nonnegative integer. Note $S = \bigcup_{k=1}^{\alpha} S_{k,m}$, $m = 1, 2, \ldots, q(\ell, k)$.

Let us further consider multisplittings of the form

$$\{D - L_{\ell,k,m}, U_{\ell,k,m}, E_k\}_{k=1}^{\alpha}, \text{ where } L_{\ell,k,m} = L_{k,m} \equiv \begin{cases} -a_{ij},\ j < i,\ i,j \in S_{k,m} \\ 0 \text{ otherwise,} \end{cases} \tag{21}$$

and $U_{\ell,k,m} = U_{k,m}$, for all $\ell = 1, 2, \ldots$. The $n \times n$ nonnegative diagonal matrices E_k, $1 \le k \le \alpha$, are defined such that their ith diagonal entry $(E_k)_{ii}$ is calculated as follows

$$(E_k)_{ii} = \begin{cases} 1 & \text{if } i \in S_{k,1} \text{ and } i \notin S_{j,1},\ j \ne k, \\ 0.5 & \text{if } i \in S_{k,1} \cap S_{k-1,1} \text{ or } i \in S_{k,1} \cap S_{k+1,1}, \\ 0 & \text{if } i \notin S_{k,1}. \end{cases}$$

Experiments were performed with almost linear systems of different orders. The conclusions were similar for all tested problems. Here we discuss the results obtained with $d = 64$ and $d = 200$, that originate almost linear systems of sizes 4096 and 40000 respectively. In this paper all the times obtained for the parallel algorithms correspond to REAL times; moreover, they are reported in seconds. The initial vector used was $x^{(0)} = (1, \ldots, 1)^T$. The stopping criterion used for the almost linear system of size 4096 was $\|x^{(\ell)} - v\|_2 \le h^2$, where $\|\cdot\|_2$ is the Euclidean norm and v is the vector which entries are the values of the exact solution of (18) on the nodes (ih, jh), $i, j = 1, \ldots, d$. However, for the almost linear system of size 40000 the convergence criterion was changed to $\|x^{(\ell)} - x^{(\ell-1)}\|_1 < 10^{-5}$.

On the other hand, to solve the one-dimensional nonlinear equation (3) the Newton method is used. The best results were obtained performing only one iteration of this method.

Table 1 shows some of these results for the almost linear system of size 4096, setting $\omega = \mu = 1$ in Algorithm 1 and using different multisplittings

depending on the number of processors used (α) and on the choice of the values n_k, $1 \leq k \leq \alpha$. Moreover, in this table, the case in which the splittings do not change with the local iterations (i.e., $L_{k,m} = L_{k,1}$, $m = 1, 2, \ldots, q(\ell, k)$) is analyzed together with the case in which the splittings change according to (19) and (21). No overlapping is considered, that is, the integer b in (20) is taken as zero. It is observed that when the splittings change, the number of global iterations decreases. Therefore, the communications among processors are reduced and less execution time is observed.

		Without varying the splittings			Varying the splittings		
α / n_k	$q(\ell,k)$	It.	Time Cluster	Time SP2	It.	Time Cluster	Time SP2
2	1,1	4292	70.46	11.82	4292	70.46	11.82
2048	2,2	2161	64.21	10.53	2144	59.42	10.29
2048	4,4	1091	58.11	10.14	1065	53.79	9.54
	8,8	552	58.74	10.36	532	53.69	9.85
	3,2	1802	70.45	11.81	1786	68.72	11.89
2	1,1	4266	111.4	15.95	4266	111.4	15.95
1216	3,3	1435	85.84	14.11	1416	83.57	13.52
2880	8,8	545	81.55	14.16	530	70.71	12.83
	10,9	476	80.04	13.98	464	69.35	12.60
4	1,1,1,1	4391	57.82	7.73	4391	57.82	7.73
1024	3,3,3,3	1499	39.50	5.98	1447	37.33	5.92
1024	4,4,4,4	1113	34.94	6.01	1081	35.12	5.94
1024	3,4,4,3	1206	38.79	6.46	1152	36.14	6.30
1024	8,8,8,8	580	40.08	6.82	536	36.57	6.64
4	1,1,1,1	4418	62.84	7.77	4418	62.84	7.77
1216	3,3,3,3	1513	44.27	6.55	1453	41.89	6.42
832	4,4,4,4	1145	42.38	6.51	1085	39.49	6.30
832	3,4,4,3	1256	36.94	5.81	1195	34.52	5.62
1216	4,5,5,4	989	36.69	5.89	930	33.89	5.61

Table 1. Non-stationary synchronous models without overlap. Size of the almost linear system: 4096.

One can observe that the number of iterations of the non-stationary algorithms decreases when the parameters $q(\ell, k)$ are increased. Furthermore, if the decrease in the number of global iterations balances the realization of more local updates then, less execution time is observed. Note that when $q(\ell, k) = 1$ and the splittings do not change with the local iterations, the method reduces to the parallel nonlinear Gauss-Seidel method (see [18]) and as it can be appreciated the non-stationary parallel methods are generally better than the parallel nonlinear Gauss-Seidel method.

On the other hand, it is interesting to compare the parallel results of Table 1 with the results of the well-known one-step Gauss-Seidel Newton method [13]. The latter performs 4196 iterations and the CPU time in the IBM RS/6000 SP computer was 20.09 seconds. So, we calculated the speed-up setting as sequential reference algorithm that method, that is, we have considered Speed-up= $\frac{\text{CPU time of one-step GS Newton algorithm}}{\text{REAL time of parallel algorithm}}$. In this context, it is observed that we obtain parallel non-stationary algorithms such that processors can achieve

between 84 % and 105 % of efficiency ($\frac{\text{Speed-up}}{\text{processors's number}}$) when it uses two processors and between 62 % and 89 % of efficiency using four processors.

			Without varying the splittings		Varying the splittings	
α	b	$q(\ell,k)$	It.	Time SP2	It.	Time SP2
n_k						
2	1	1,1	4253	12.43	4292	12.43
2048		2,2	2134	10.95	2124	10.70
2048		4,4	1072	10.42	1059	9.99
		8,8	541	10.55	530	10.22
		3,2	1777	12.35	1769	12.42
2	2	1,1	4246	12.89	4246	12.89
2048		2,2	2129	11.31	2121	11.10
2048		4,4	1068	10.67	1058	10.32
		8,8	538	10.81	529	10.49
		3,2	1769	12.67	1761	12.81
2	3	1,1	4244	13.27	4244	13.27
2048		2,2	2127	11.67	2120	11.47
2048		4,4	1067	11.00	1057	10.64
		8,8	537	11.17	529	10.78
		3,2	1763	12.99	1755	13.12
4	1	1,1,1,1	4327	9.05	4327	9.05
1216		3,3,3,3	1461	7.04	1432	6.89
832		4,4,4,4	1102	6.87	1071	6.76
832		3,4,4,3	1210	6.36	1179	6.29
1216		4,5,5,4	951	6.31	920	6.23
4	2	1,1,1,1	4311	9.84	4311	9.84
1216		3,3,3,3	1452	7.37	1427	7.35
832		4,4,4,4	1093	7.19	1068	7.12
832		3,4,4,3	1200	7.01	1175	6.96
1216		4,5,5,4	942	6.95	917	6.92

Table 2. Non-stationary synchronous models with overlap. Size of the almost linear system: 4096.

The above conclusions are independent of the computer used. However, the network of the cluster is very slow compared to the network of the other computing platform. Hence, while the REAL time and the CPU time are similar in the IBM RS/6000 SP computer, there is a significant difference between these two times in the cluster. In the rest of this section all the numerical experiments have been run in the IBM RS/6000 SP multiprocessor.

Table 2 illustrates the influence of the overlap according to different choices of the overlapping level $b = 1, 2, 3$; see (20). Note that the parameter b indicates that the splitting assigned to a processor k, $k = 1, 2, \ldots, \alpha$, has an overlap of $2b$ blocks (each one of size d) with the splittings assigned to the processors $k-1$ and $k+1$. The conclusions are similar to those presented in Table 1. However, it is observed that while the number of iterations decreases when the overlap increasesle, this decrease does not get less execution time. This is due to the increase of the number of operations performed at each processor and the increase of the communications among processors.

Now, we report in Table 3 results of non-stationary methods for the almost linear system of size 40000. Moreover, we have calculated for each method, the error $||x^{(\ell)} - v||_2$. As it can be appreciated when the non-stationary parameters

α	b / n_k	$q(\ell,k)$	Without varying the splittings: It.	Time SP2	Error	Varying the splittings: It.	Time SP2	Error
4	0	1,1,1,1	40983	818.18	0.00025	40983	818.18	0.00025
	$n_k = 10000$	3,3,3,3	14961	611.29	0.00012	14799	604.55	0.00012
	$1 \le k \le 4$	4,4,4,4	11484	598.31	0.00010	11318	585.85	0.00010
4	1	1,1,1,1	40757	845.37	0.00025	40757	845.37	0.00025
	$n_k = 10000$	3,3,3,3	14820	622.18	0.00012	14741	616.90	0.00012
	$1 \le k \le 4$	4,4,4,4	11365	607.84	0.00010	11280	606.88	0.00010
6	0	1,1,1,1,1,1	41266	608.52	0.00025	41266	608.52	0.00025
$n_k = 6800, k = 1,6$		3,3,3,3,3,3	15120	446.33	0.00012	14874	437.73	0.00012
$n_k = 6600$		5,5,5,5,5,5	9474	433.37	0.000095	9223	415.58	0.000094
$2 \le k \le 5$		4,5,5,5,5,4	9622	430.46	0.000096	9369	413.59	0.000095
6	1	1,1,1,1,1,1	40925	636.56	0.00025	40925	636.56	0.00025
$n_k = 6800, k = 1,6$		3,3,3,3,3,3	14906	463.43	0.00012	14786	453.39	0.00015
$n_k = 6600$		5,5,5,5,5,5	9318	440.76	0.000094	9183	434.38	0.000094
$2 \le k \le 5$		3,4,4,4,4,3	11679	449.56	0.00010	11548	445.59	0.00010

Table 3. Non-stationary synchronous models without and with overlap. Size of the almost linear system: 40000.

$q(\ell,k)$ increase the error is reduced and therefore the approximation to the solution of the semilinear elliptic partial differential equation (18) is better. The remaining conclusions are similar to those obtained for the almost linear system of size 4096.

Figure 1 illustrates the influence of the parameters $\omega = \mu \neq 1$ for different overlapping levels, $b = 0, 1, 2$ when Algorithm 1, varying the splittings, is used. These results correspond to the problem of size 4096 when we use four processors and the multisplitting is defined by $n_k = 1024$, $k = 1, 2, 3, 4$. As it can be observed, in a neighborhood of the optimum relaxation parameter ω the models with overlap are better than the corresponding non overlapped one. We note that similar results were obtained without varying the splittings.

To finish this section we consider two different implementations of asynchronous Algorithm 2. In the first implementation we consider α processors connected to a host processor, where α is the number of splittings. Thus, we use $\alpha + 1$ processors. The role of the host processor is to receive, in an asynchronous way, the approximation computed by other processors, to update the global approximation and to send it to the corresponding processor.

In the second implementation we use α processors, as many as splittings. Then, one of the processors, we assume the first one, has to compute the approximation corresponding to one of the splittings and, moreover, it has to take the role of host processor. For this purpose, in the process executed by this processor we have intercalated some PVMe calls between the sentences which compute its approximation. This allows us to check if the approximations of some of the other processors have arrived. In this case the host processor executes the following tasks,

1. it stops the calculation of its approximation and it receives the approximation or approximations sent by other processors,
2. it updates the global approximation,

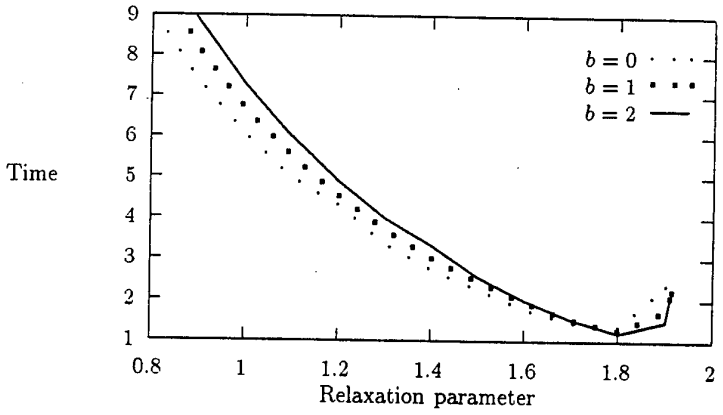


Fig. 1. Comparison synchronous parallel models for different overlapping levels. Non-stationary parameters $q(\ell, k) = 4$, $\ell = 1, 2, \ldots, k = 1, 2, 3, 4$.

3. if the stopping criterion is not satisfied, then it sends the updated approximation to the corresponding processors, and finally
4. it continues computing its approximation.

Note that in this implementation there are some waiting times.

Figures 2 and 3 illustrate, respectively, the behavior of the above asynchronous implementations of Algorithm 2. In these figures, the multisplitting 1 makes reference to the one obtained from $n_k = 1024$, $k = 1, 2, 3, 4$, while the multisplitting 2 corresponds to the values $n_1 = n_4 = 1216$, $n_2 = n_3 = 832$. Overlap and variation of the splittings are not considered in these figures. The conclusions were similar to those of the synchronous models whether the overlap and variation of the splittings were considered or not. However, these asynchronous implementations have not accelerated the convergence. This is due to the fact that in the asynchronous implementations of our example the communications increase compared to the synchronous ones, while the number of operations performed remains of the same order. This is specially problematic when a distributed memory multiprocessor is used.

References

1. Bai, Z.: Parallel nonlinear AOR method and its convergence. Computers and Mathematics with Applications, Vol. 31(2) (1996) 21–31
2. Baudet, G. M.: Asynchronous iterative methods for multiprocessors. Journal of the Association for Computing Machinery, Vol. 25(2) (1978) 226–244

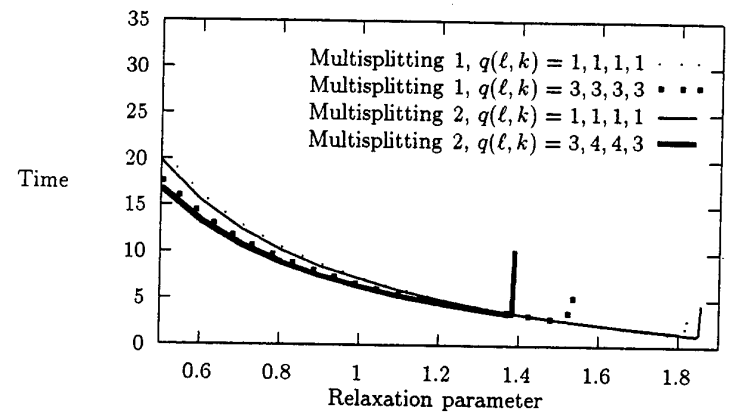


Fig. 2. Comparison asynchronous parallel models without varying the splittings and without overlap (first implementation). Size of almost linear system: 4096.

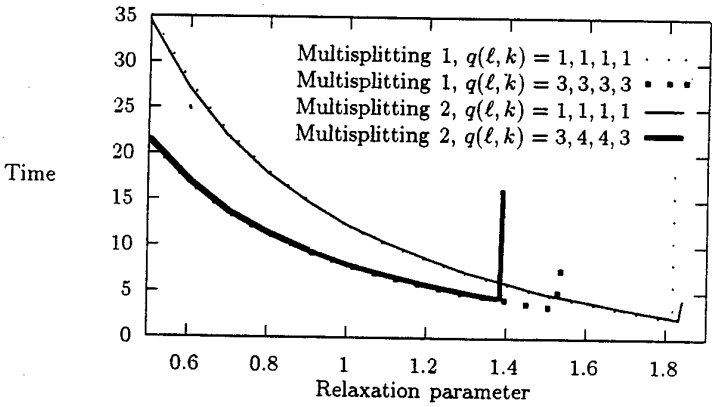


Fig. 3. Comparison asynchronous parallel models without varying the splittings and without overlap (second implementation). Size of almost linear system: 4096.

3. Berman A., Plemmons R. J.: Nonnegative Matrices in the Mathematical Sciences. Academic Press, New York, third edition (1979). Reprinted by SIAM, Philadelphia (1994)
4. Bertsekas, D. P., Tsitsiklis, J. N.: Parallel and Distributed Computation. Prentice-Hall, Englewood Cliffs, New Jersey (1989)
5. Birkhoff, G.: Numerical Solution of Elliptic Equations. Vol. 1 of CBMS Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, Philadelphia (1970)
6. Bru R., Elsner L., Neumann M.: Models of parallel chaotic iteration methods. Linear Algebra and its Applications, Vol. 103 (1988) 175–192
7. Frommer, A.: Parallel nonlinear multisplitting methods. Numerische Mathematik, Vol. 56 (1989) 269–282
8. Fuster, R., Migallón, V., Penadés, J.: Non-stationary parallel multisplitting AOR methods. Electronic Transactions on Numerical Analysis, Vol. 4 (1996) 1–13
9. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V.: PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187. Oak Ridge National Laboratory, Tennessee (1994)
10. IBM Corporation: IBM PVMe for AIX User's Guide and Subroutine Reference. Technical Report GC23-3884-00, IBM Corp. Poughkeepsie, New York (1995)
11. Mas, J., Migallón, V., Penadés, J., Szyld, D B.: Non-stationary parallel relaxed multisplitting methods. Linear Algebra and its Applications, Vol. 241/243 (1996) 733–748
12. O'Leary, D. P., White, R. E.: Multi-splittings of matrices and parallel solution of linear systems. SIAM Journal on Algebraic Discrete Methods, Vol. 6 (1985) 630–640
13. Ortega, J. M., Rheinboldt, W. C.: Iterative Solution of Nonlinear Equations in Several Variables. Academic Press, San Diego (1970)
14. Ostrowski, A. M.: Über die determinanten mit überwiegender hauptdiagonale. Commentarii Mathematici Helvetici, Vol. 10 (1937) 69–96
15. Robert, F., Charnay, M., Musy, F.: Itérations chaotiques série-parallèle pour des équations non-linéaires de point fixe. Aplikace Matematiky, Vol. 20 (1975) 1–38
16. Sherman, A.: On Newton-iterative methods for the solution of systems of nonlinear equations. SIAM Journal on Numerical Analysis, Vol. 15 (1978) 755–771
17. Varga, R. S.: Matrix Iterative Analysis. Prentice Hall, (1962)
18. White, R. E.: Parallel algorithms for nonlinear problems. SIAM Journal on Algebraic Discrete Methods, Vol. 7 (1986) 137–149

Spatial Data Locality With Respect to Degree of Parallelism in Processor-And-Memory Hierarchies

Renato J. O. Figueiredo[1], José A. B. Fortes[1] and Zina Ben Miled[2]

[1] School of ECE - Purdue University, West Lafayette, IN 47907
[2] Department of EE - Purdue University, Indianapolis, IN 46202
{figueire,fortes}@ecn.purdue.edu, miled@engr.iupui.edu

Abstract. A system organized as a Hierarchy of Processor-And-Memory (HPAM) extends the familiar notion of memory hierarchy by including processors with different performance in different levels of the hierarchy. Tasks are assigned to different hierarchy levels according to their degree of parallelism. This paper studies the spatial locality (with respect to degree of parallelism) behavior of simulated parallelized benchmarks in multi-level HPAM systems, and presents an inter-level cache coherence protocol that supports inclusion and multiple block sizes on an HPAM architecture. Inter-level miss rates and traffic simulation results show that the use of multiple data transfer sizes (as opposed to a unique size) across the HPAM hierarchy allows the reduction of data traffic between the uppermost levels in the hierarchy while not degrading the miss rate in the lowest level.

1 Introduction

The Hierarchical Processor-And-Memory (HPAM) architecture [15] has been proposed as a cost/effective approach to parallel processing. The HPAM concept is based on a heterogeneous, hierarchical organization of resources that is similar to conventional memory hierarchies. However, each level of the hierarchy has not only storage but also processing capabilities. Assuming that the top (i.e. first) level of the hierarchy is the fastest, any given memory level is extended with processors that are slower, less expensive and in larger number than those in the preceding level. Figure 1 depicts a generic 3-level HPAM machine.

The mapping of an application to an HPAM system is based on the degrees of parallelism that the application exhibits during its execution. Each level of an HPAM hierarchy handles portions of code whose parallelism degree is within a certain range. Levels with large number of slow processors and large memory capacity (bottom levels) are responsible for the highly parallel fractions of an application, whereas levels with small number of fast processors and memories are responsible for the execution of sequential and moderately parallel code.

An HPAM machine exploits heterogeneity, computing-in-memory and locality of memory references with respect to degree of parallelism to provide superior cost/performance over conventional homogeneous multiprocessors. Previous

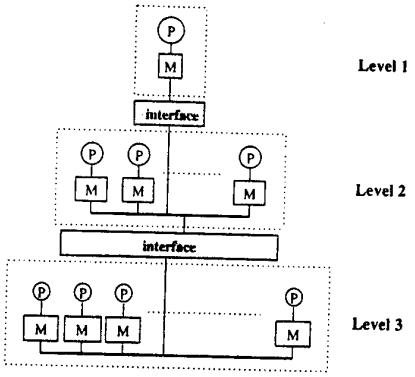


Fig. 1. Processor and memory organization of a 3-level HPAM

studies [14, 15] have empirically established that applications exhibit temporal locality with respect to degree of parallelism. This paper extends these studies by empirically establishing that applications also exhibit *spatial* data locality. Furthermore, this paper studies the impact of using multiple transfer sizes across an HPAM hierarchy with more that two levels in inter-level miss rates and traffic. To this end, this paper proposes an inter-level coherence protocol that supports inclusion and multiple block sizes across the hierarchy.

The data locality studies have been performed through execution-driven simulation of parallelized benchmarks. Several benchmarks from three different suites (CMU [9], Perfect Club [7] and Spec95 [11]) have been instrumented to detect do-loop parallelism with the Polaris [12] parallel compiler. The stream of memory references generated by a benchmark is simulated by a multi-level memory hierarchy simulator that implements the proposed inter-level coherence protocol to obtain measurements of data locality, data traffic and invalidation traffic at each HPAM level.

The rest of this paper is organized as follows. Section 2 introduces the HPAM machine model and discusses coherence protocols for an HPAM architecture. Section 3 presents the methodology used to perform the data locality studies, and Section 4 presents simulation results and analysis of the locality behavior of applications and of the proposed inter-level coherence protocol. Section 5 concludes the paper.

2 HPAM Architecture

A hierarchical processor-and-memory (HPAM) machine is a heterogeneous, multilevel parallel computer. Each HPAM level contains processors, memory and interconnection network. The speed and number of processors, latency and capacity of memories and network differ between levels in the hierarchy. The following characteristics hold across the different HPAM levels from top to bottom:

individual processor performance decreases, number of processors increases, and memory/network latency and capacity increase.

Tasks are assigned to HPAM levels according to their *degree of parallelism* (DoP). Highly parallel code fractions of an application are assigned to bottom levels, where large number of processors and large memory capacity are available, while sequential and modestly parallel fractions are assigned to top levels, where a small number of fast processors and memories are available.

The HPAM approach to computing-in-memory bears similarities with IRAM and PIM efforts [2, 10], but it relies on heterogeneity and locality with respect to degree of parallelism to build a hierarchy of processor-and-memory subsystems where each subsystem is designed to be cost-efficient in its parallelism range. A massively parallel system implemented with dense, relatively inexpensive and slow memory technology, can be very efficient for highly parallel code, while a tightly-coupled symmetric multiprocessor containing a smaller amount of fast, expensive memory, can be very efficient for code mostly sequential or with moderate parallelism. The merging of these systems under the HPAM concept provides a cost-efficient solution for applications with different levels of parallelism.

2.1 Inter-Level Coherence Protocols

A distributed shared-memory (DSM) implementation of HPAM is assumed in this paper. Each level of such shared-memory HPAM machine relies on caching of data from remote levels to reduce inter-level bandwidth requirements and improve remote access latency. Therefore, cache coherence has to be enforced both inside an HPAM level and among different levels. Cache-coherence solutions that use a combination of different protocols (snoopy and directory-based) have been proposed and implemented [6] for homogeneous DSMs, and can be reused in an HPAM context. However, an HPAM machine can take advantage of coherence solutions that exploit its heterogeneous nature.

In this paper, the potential advantages of having multiple line sizes across the hierarchy are studied. Similar to conventional uniprocessor memory hierarchies, multiple line sizes in an HPAM context can provide low miss rates in the bottom levels of the hierarchy while not sacrificing traffic and miss penalty in the upper hierarchy levels. In this section, a coherence protocol that allows requests to be generated by any level of the hierarchy (as opposed to a conventional memory hierarchy, where all accesses are generated in the topmost level), and supports both inclusion and multiple line sizes, is described. Similar to the MESI coherence protocol [8], the protocol assigns one of four states to each memory block and relies on invalidations to maintain coherence.

Let the hierarchical organization have h levels, where level 1 is the top level and h is the bottom hierarchy level. Let ls_i be the line size (also referred to as block size) in level i. All data transfers between adjacent levels i and $i+1$ have size ls_i. Let $B_j(i)$ be block i in level j, where $i \geq 0$ and $1 \leq j \leq h$. Assume that block sizes across the hierarchy satisfy the relation:

$$\frac{ls_j}{ls_k} = 2^x, j \geq k, x \in N \quad (1)$$

Given this alignment, let the $\frac{ls_j}{ls_k}$ *sub-blocks* in level k of a block $B_j(i)$ be defined as:

$$B_k\left(\frac{ls_j}{ls_k} * i\right), B_k\left(\frac{ls_j}{ls_k} * i + 1\right), ..., B_k\left(\frac{ls_j}{ls_k} * i + \frac{ls_j}{ls_k} - 1\right), j \geq k \quad (2)$$

and the unique *superblock* in level l of $B_j(i)$, $l \geq j$, be defined as:

$$B_l\left(\lfloor \frac{ls_j}{ls_l} * i \rfloor\right) \quad (3)$$

For the proposed inter-level coherence protocol, blocks can be in any of the following four states:

- **Invalid (I)**: data in the block is entirely non-valid
- **Accessible (A)**: data in the block is valid and may be shared (read-only) by one or more processors
- **Reserved (R)**: data in the block is the only valid copy in the hierarchy
- **Partially Invalid (P)**: at least one sub-block of the memory block is outdated (due to a write in an upper-level memory)

Memory access operations consist of read and write commands that can be issued from any level j of the hierarchy. These operations are considered atomic. The read/write commands are defined in terms of four primitive coherence operations, as follows: (the algorithms used to implement these basic inter-level coherence operations are defined in Appendix A)

- **ULI**($B_j(i)$)(**U**pper-**L**evel **I**nvalidate): Invalidates all sub-blocks of $B_j(i)$
- **ULW**($B_j(i)$)(**U**pper-**L**evel **W**riteback): Writes back dirty data to $B_j(i)$ from upper-level sub-blocks; sets sub-blocks to *Accessible* (read-only)
- **LLP**($B_j(i)$)(**L**ower-**L**evel **P**artial-Invalidate): Sets all superblocks of $B_j(i)$ *Partially Invalid*
- **LLR**($B_j(i)$)(**L**ower-**L**evel **R**ead): Fetches block $B_j(i)$ from lower-level superblocks; sets super-blocks to *Accessible* (read-only)

The coherence protocol implements read/write operations as combinations of these four primitives. In order to allow for multi-level inclusion, i.e., (definition here), the coherence protocol enforces the following properties:

1. If a block $B_j(i)$ is *Partially Invalid*, then all of its superblocks must also be *Partially Invalid*
2. If a block $B_j(i)$ is *Invalid*, then all of its sub-blocks must also be *Invalid*
3. If a block $B_j(i)$ is *Reserved*, then all of its sub-blocks must be *Invalid* and all of its superblocks must be *Partially Invalid*
4. If a block $B_j(i)$ is *Accessible*, then all of its sub-blocks are either *Invalid* or *Accessible*, and all of its superblocks are either *Partially Invalid* or *Accessible*

Algorithms for the coherent write and read operations of a block $B_j(i)$ in level j are presented in Appendix A. Property **1** allows the coherence controller to fetch the most recent copy of a block $B_j(i)$ if any sub-block of it has been modified by an upper-level processor before completing a read or write request. Property **3** ensures that a processor can complete a write to block $B_j(i)$ when the state of $B_j(i)$ is *Reserved* without involving other processors of the write, since it is the only processor that has a valid copy of the block.

An example of the inter-level coherence protocol operation on a 3-level configuration is shown in Figure 2. Each memory block is represented in this figure by both its state (gray-shaded boxes) and contents of each of its sub-blocks. Note that the block sizes differ among the levels; a block in level 2 is twice larger than a block in level 1 and four times larger than a block in level 0.

The example begins with the configuration of Figure 2(a): the bottom level has valid data in the *Reserved* state, and the other levels have invalid data. Level 0 then issues a memory *read* of block $B_0(1)$. The protocol issues a *lower-level read* primitive, bringing a sub-block of level 2 to level 1, containing values **x** and **y**, then a subblock of level 1 to level 0, containing the desired data (**y**). All blocks involved in this transaction become *Accessible* (Figure 2(b)).

Suppose level 1 issues a *write* to block $B_1(0)$ and let **t** and **u** denote the new contents of the respective sub-blocks. The protocol handles this request by invalidating upper level sub-blocks ($B_0(0)$ and $B_0(1)$), setting the lower level superblock $B_2(0)$ to *Partially Invalid* and setting the state of $B_1(0)$ to *Reserved* (Figure 2(c)).

The next memory reference in this example is a *read* of block $B_0(3)$ (Figure 2(c)). Similarly to the first read, data is brought from level 2 to level 1, then to level 0. The states of the blocks in levels 0 and 1 become *Accessible*. However, the state of the block $B_2(0)$ in level 2 remains *Partially Invalid* to flag that at least one of its sub-blocks ($B_1(0)$ in this case) contains data that needs to be written back, as Figure 2(d) shows.

The last memory reference of this example is a *read* of $B_2(0)$. This reference generates a write-back request to the *Reserved* sub-block $B_1(0)$ in the upper level. The *Reserved* block in level 1 becomes *Accessible* (Figure 2(e)). This assumes the existence of state bits associated with each sub-block of the adjacent upper level. The implementation of read and write fences for relaxed consistency models requires that the coherence protocol handles acknowledgments of all messages exchanged between adjacent levels (not shown in the primitives of Appendix A). The completion of an operation (read or write) issued by processor P with respect to all other processors is signalled by the arrival of positive acknowledgment from upper and lower adjacent levels.

3 Simulation Models and Methodology

The locality studies presented in this paper are based on a generic four-level HPAM architecture. Each level is labelled from *0* to *3*, where *0* represents the

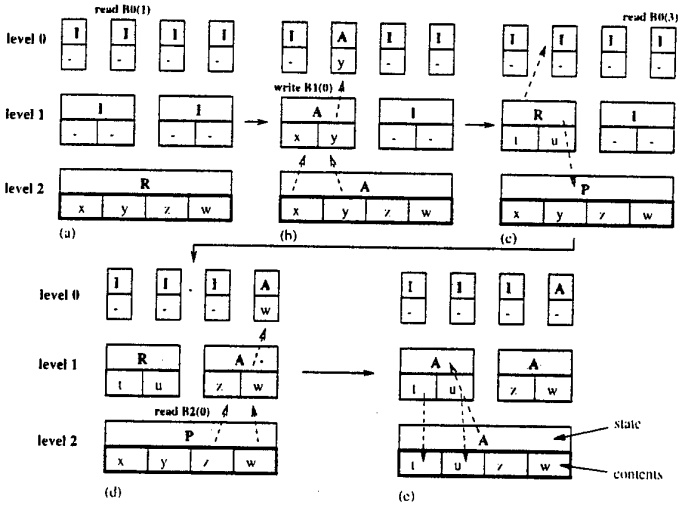


Fig. 2. Example of coherence protocol **I** for 3-level configuration. Line states are shown in the top of each box: P (Partially Invalid), I (Invalid), A (Accessible) and R (Reserved) and contents of blocks are shown below the state.

topmost level (smallest degree of parallelism). Level i in the hierarchy is responsible for executing fractions of code which have degree of parallelism greater or equal to DoP_i and less than DoP_{i+1} (or infinity for the bottom level). The degrees of parallelism in this study were fixed as powers of ten, $DoP_0 = 1$, $DoP_1 = 10$, $DoP_2 = 100$ and $DoP_3 = 1000$. The variables used in the locality experiments in this paper are defined as follows:

- Line size of level i (ls_i): represented in log_2 notation in this paper (i.e. $ls_i = 8$ means a line size of $2^8 = 256$ bytes). This parameter determines the amount of data that is transferred to level i when a data miss occurs in this level. It is assumed that each level has an inter-level cache that is shared by all level processors. The shared caches at each level are assumed to be ideal (fully associative and infinite) in the simulations performed.
- Data miss rate at level i (mr_i): percentage of memory references (loads and stores) in level i that miss. Misses in level i can be serviced either by disk (cold misses) or by another level j, $i \neq j$. The latter case will be referred to as *inter-level misses* throughout this paper. Intra-level misses, which are serviced by processors in the same level, are not modeled.
- Data traffic between levels i and $i+1$ ($tr_{i,i+1}$): aggregate amount of data transferred between levels i and $i+1$. Inter-level communication is assumed to occur between adjacent levels only. Therefore, $tr_{i,i+1}$ accounts not only for the amount of data exchanged between levels i and $i+1$, but also for any data transfer from/to level j, $j \leq i$ to/from level k, $k \geq i+1$.

The simulation methodology combines compiler-assisted parallelism identification with execution-driven simulation of benchmarks. An application under study is first instrumented with the Polaris [12] source-to-source parallelizing compiler to detect do-loop parallelism. The instrumented Fortran code has tags inserted in the beginning and end of each loop that indicate the degree of parallelism of the loop. The Polaris-generated code is then compiled, and the executable code is used as input to an execution-driven simulator. The simulator models a multi-level, shared-memory hierarchy, and is built on top of Shade [1].

The simulator engine traces each memory data access during program execution. The engine identifies the level that issues each access by comparing the current degree-of-parallelism tag (inserted in the instrumentation phase) with the parallelism thresholds DoP_i. The memory access is forwarded to the appropriate level cache handler, which characterizes the access either as a hit (data has previously been in the level's cache) or a miss (either the data is present in another level's cache or needs to be fetched from disk). Coherence messages are sent by the cache handler to other level caches on misses, according to the cache coherence protocol under use. Hence, the inter-level coherence protocol behavior is modeled. However, the intra-level coherence protocol is not modeled. The miss rate and traffic results obtained with such model are therefore optimistic, since ideal caches and intra-level sharing are assumed. Nonetheless, this simplified model is able to capture the inherent spatial locality with respect to degree of parallelism of the applications under study. Miss rates degrade when finite caches and intra-level sharing are considered, but locality with respect to degree of parallelism is still evident for non-ideal memory systems [3].

The following benchmarks from the CMU, Perfect Club and Spec95 suites have been used in the spatial locality studies: Radar, Stereo, FFT2 and Airshed (CMU Parallel Suite [9]); TRFD, FLO52, ARC2D, OCEAN and MDG (Perfect Club Suite [7]); Hydro2d and Swim (Spec95 Suite).

4 Simulation Results and Analysis

For each benchmark, simulations have been performed for various line sizes (ls_i), and measurements of miss rates (mr_i) and data traffic ($tr_{i,i+1}$) have been collected. Two inter-level coherence protocols have been considered in this study. Initially, a homogeneous solution analogous to a cache-only (COMA) protocol [4] is used to observe the inherent locality behavior of the set of benchmarks under study. In this scenario, a block can migrate to any level of the hierarchy, i.e., there is no fixed home node associated with a given memory block. Such scenario is referred to as "migration protocol" in the rest of this paper. The other coherence solution considered is the heterogeneous protocol introduced in Section 2.1. Such scenario is referred to as "inclusion protocol" in the rest of this paper.

Subsection 4.1 presents data obtained when a migration protocol with unique line sizes across the hierarchy is assumed. Such scenario is used as a basis for the analysis of (level) data locality. Subsection 4.2 compares the results obtained from the migration scenario to results obtained when the coherence protocol en-

forcing inclusion presented in Section 2.1 is used and line sizes are allowed to be different across levels.

4.1 Migration protocol with unique line size across levels

The results obtained for this protocol configuration confirm empirically that applications have good spatial locality with respect to degree of parallelism, in addition to previously observed [14] temporal locality with respect to degree of parallelism. Inter-level miss rates are low for small line sizes: a highest miss rate of 17.3% occurs in *Swim* for 16-Byte line sizes, but typical values are around 1%. Furthermore, inter-level miss rates tend to decrease as the line size gets larger.

Figure 3 shows this trend for the benchmark *FLO52*, assuming a 4-level HPAM. The figure is divided into four sub-plots, each corresponding to an HPAM level, labelled *lvl0* through *lvl3* in the x-axis. Each sub-plot is further divided into six line sizes, ranging from $ls = 2^4$ to $ls = 2^{14}$ bytes. For each level and line size, the *absolute number* of misses is plotted in the y-axis in log scale, and the corresponding *miss rate* is indicated in the x-axis between parentheses. The different shades of the bars in the y-axis correspond to the *percentage* of inter-level misses that are either cold misses or serviced by another level. To illustrate this notation, consider the case where FLO52 runs on a 4-level HPAM with line size of $2^6 = 64$ bytes in all levels. The inter-level miss rates for levels 0 through 3 are 3.40%, 0.35%, 0.20% and 0.63%. For level *lvl2* and line size of 64 bytes, approximately half of the inter-level misses are serviced by level 3, 20% are serviced by level 1, 30% by level 0, and a negligible fraction is due to cold misses.

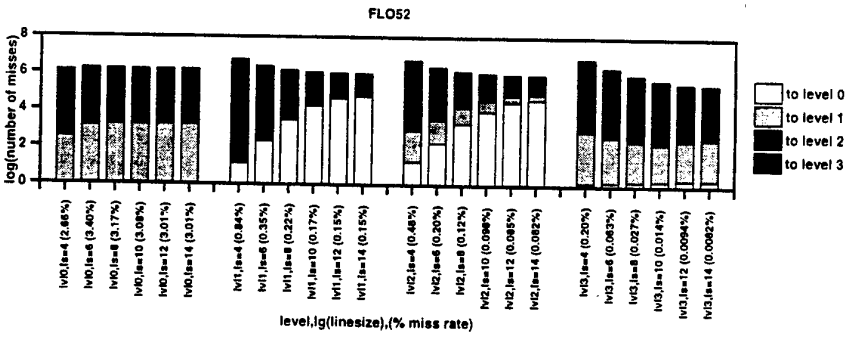


Fig. 3. Misses: FLO52, 4 levels, protocol **C**

Figure 3 shows that the spatial locality behavior for *FLO52* varies across hierarchy levels. For level 0, the inter-level miss rate remains approximately constant, slightly degrading as the line size increases. In contrast, the miss rate decreases about two orders of magnitude as line size increases from 2^4 to 2^{14} in

level 3. Such behavior suggests that the parallel fraction of FLO52 that executes in the lowest hierarchy level operates on large, regular data structures that benefit from fetching large data lines on a miss.

While a larger line size tends to improve inter-level miss rates, it also tends to increase inter-level data traffic. Figure 4 shows how data traffic between adjacent levels varies as a function of line size for the benchmark FLO52. Notice that the traffic between levels 0 and 1 in this case increases by about three orders of magnitude for the range of line sizes considered, while the traffic between levels 2 and 3 increases only by about two orders of magnitude across the same line size range. Such behavior can be explained with the aid of the inter-level miss rate profile for FLO52 (Figure 3). The larger line sizes brought to levels 2 and 3 often contain data that is likely to be used in future references, while larger line sizes in levels 0 and 1 most often bring data that remains unused. Since traffic is proportional to the product of number of misses and line size, if the number of misses do not decrease as line size increases, the traffic increases.

VECPAR'98 - 3rd International Meeting on Vector and Parallel Processing

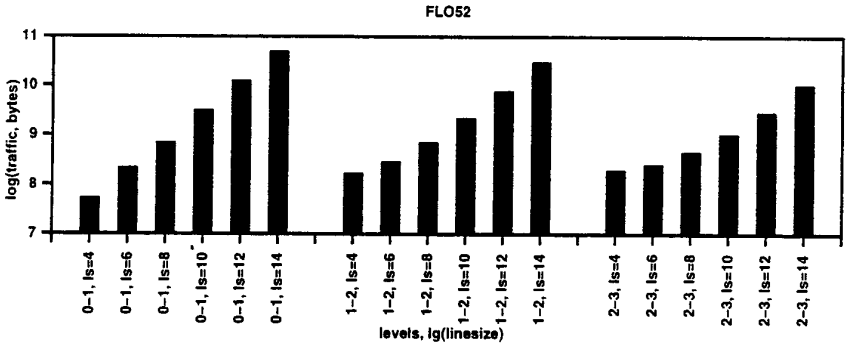


Fig. 4. Traffic: FLO52, 4 levels, protocol **C**

Table 1 summarizes the maximum and minimum miss rates found for each benchmark studied, as line size varies from 2^4 to 2^{14} bytes in an HPAM organization with four levels. The benchmarks Stereo and Swim have only three distinct levels of parallelism detected by Polaris, and FFT2 has only two. For these benchmarks, the HPAM levels that do not generate memory references have null minimum and maximum miss rate entries in Table 1.

The results summarized in Table 1 show good spatial locality with respect to degree of parallelism for the benchmarks studied. The maximum inter-level miss rate observed for the lowest level of the hierarchy is 0.9%; for the topmost level, the maximum inter-level miss rate observed is 17.3% for *Swim*, and less than or equal to 4.01% for all other benchmarks. In general, the best miss rates are found in the lowest level of the architecture, where loops with high degree of parallelism are executed.

Table 1. Min/Max inter-level miss rates for a 4-level HPAM configuration

Benchmark	Inter-level Miss rate							
	Level 0		Level 1		Level 2		Level 3	
	min	max	min	max	min	max	min	max
FLO52	2.66%	3.40%	0.15%	0.84%	0.082%	0.48%	0.0082%	0.20%
TRFD	0.12%	1.30%	5.28%	6.97%	0.87%	1.08%	0.046%	0.086%
OCEAN	0.0052%	0.17%	0.096%	1.63%	0.27%	1.70%	0.023%	0.45%
MDG	0.39%	1.50%	0.77%	3.16%	0.049%	3.76%	0.0012%	0.097%
ARC2D	1.03%	2.17%	0.0043%	1.45%	0.0071%	1.41%	0.0004%	0.062%
Airshed	2.92%	4.01%	0.48%	0.72%	0.049%	0.049%	0.018%	0.037%
Stereo	0.021%	1.48%	-	-	0.049%	1.51%	0.011%	0.12%
Radar	0.00039%	0.066%	0.27%	1.63%	0.035%	3.79%	0.0015%	0.90%
FFT2	0.00008%	0.065%	-	-	-	-	0.0050%	0.65%
Hydro2d	0.0053%	2.88%	1.78%	10.60%	0.29%	6.55%	0.00038%	0.0074%
Swim	12.81%	17.30%	0.80%	9.48%	0.0010%	0.011%	-	-

4.2 Inclusion protocol

Identical line sizes: The inclusion protocol was initially studied assuming that a unique line size is used across all HPAM levels, similar to the migration protocol discussed in Subsection 4.1. Since the level caches are assumed to be ideal, and intra-level sharing is not modelled, the results for the inclusion coherence protocol with unique line size do not differ considerably from the results obtained from the migration protocol simulations, in general. Assuming such idealized memory model, both protocols yield measurements that characterize the *inherent* sharing behavior of the benchmarks. The inter-level miss rates obtained for this scenario differ by at most 17% from the migration scenario for *TRFD*, with an average difference of 1.4% across all benchmarks.

Distinct line sizes: Conventional uniprocessor cache hierarchies typically use distinct line sizes across the cache levels; large lines are desirable in large caches to improve miss rates, while small cache lines are desirable in small caches to avoid excessive bandwidth requirements and increases in miss penalties and conflict misses [5, 13]. An HPAM machine can also benefit from distinct line sizes across levels by reducing inter-level traffic while not sacrificing inter-level miss rates.

The inter-level miss rate and traffic profiles for the benchmark *FLO52* (Figures 3 and 4) illustrate a scenario commonly observed in the simulations performed, where a large line size effectively reduces the inter-level miss rate in level 3, but unneccessarily increases the traffic between levels 0 and 1 without improving the inter-level miss rates in these levels. The inter-level inclusion coherence protocol described in Section 2.1 supports a configuration with distinct line sizes across HPAM levels; the effects of distinct line sizes on inter-level miss rates and traffic have been captured quantitatively through simulation and are

discussed in this subsection.

Line sizes have been set up such that the relationship $ls_{i+1} > ls_i$ holds for any adjacent levels $i, i+1$. Hence, levels executing highly parallel code are assigned line sizes strictly larger than levels executing moderately parallel or sequential code. Table 2 shows how inter-level miss rates and traffic for a configuration with multiple line sizes compare to configurations with unique line sizes, for the benchmark *MDG*. The first row of Table 2 shows inter-level miss rates for the multiple line size configuration. Rows 2 through 5 of the table show miss rates obtained in four different simulations with unique line sizes, each corresponding to a line size chosen for the multiple line size scenario. The remaining rows of Table 2 show the total traffic in the level boundaries for three scenarios: multiple line sizes, unique line of smallest size (2^6 Bytes), and unique line of largest size (2^{14} Bytes).

Table 2. Traffic and inter-level miss rates for MDG with multiple line sizes: 2^6, 2^8, 2^{10} and 2^{14}

	Level 0 (ls=2^6)	Level 1 (ls=2^8)	Level 2 (ls=2^{10})	Level 3 (ls=2^{14})
Miss rate (multiple)	**1.28%**	**1.38%**	**0.10%**	**0.0012%**
Miss rate (unique, 2^6)	**0.82%**	2.17%	1.00%	0.025%
Miss rate (unique, 2^8)	0.48%	**1.44%**	0.29%	0.0073%
Miss rate (unique, 2^{10})	0.56%	0.98%	**0.10%**	0.0026%
Miss rate (unique, 2^{14})	0.39%	0.77%	0.049%	**0.0012%**
	levels 0-1	levels 1-2	levels 2-3	
Traffic (multiple)	26.6GB	12.9MB	17.7MB	-
Traffic (single, ls=2^6)	15.6GB	10.8MB	11.3MB	-
Traffic (single, ls=2^{14})	1.6TB	71.8MB	127.7MB	-

Table 2 shows that the miss rate observed in the multiple line size scenario is at most 56% larger than the rate observed for the corresponding unique line size rate (values in bold face) for each level. For levels 2 and 3, in particular, the inter-level miss rates are equal for both scenarios. When inter-level traffics are compared, the multiple line size scenario demands about an order of magnitude less traffic than the unique line size scenario with the largest line size, while demanding no more than 70% more traffic than the unique line size scenario with the smallest line size. In this example, the multiple line size configuration is therefore capable of providing very low miss rates at the lowest hierarchy level without generating excessive traffic in the upper level boundaries. When a unique line size is used, either the miss rate in the lowest level or traffic in the topmost level degrades. The same motivations for using multiple line sizes across uniprocessor memory hierarchies thus apply to a hierarchy of processor-and-memories: maintaining good miss rates across the hierarchy while avoiding the generation of unnecessary traffic in the upper levels.

Table 3 shows the average increase in the inter-level miss rate of the multiple line size configuration compared to the miss rate of a corresponding unique line size configuration for simulations performed in six of the studied benchmarks. The inter-level miss rates of the lowest levels in the hierarchy remain unchanged, with respect to the unique line size scenario, when multiple line sizes are used. Miss rates at the topmost level increase by 31% in average.

Table 3. Average ratio: miss_rate(multi)/miss_rate(single)

	Level 0	Level 1	Level 2	Level 3
Average mr_{multi}/mr_{single}	1.31	1.05	1.01	1.00

5 Conclusions

The conclusions reached in this paper provide guidelines to the design of the memory and network subsystems of an HPAM machine. The implementation of a coherence controller that supports multiple line sizes across the hierarchy is an ongoing research subject. The inclusion coherence protocol presented in this paper has been used as a proof of concept to study the advantages of fetching larger blocks of data to lower levels of the hierarchy as a means of increasing spatial locality without sacrificing traffic in the upper levels of the hierarchy. One solution under investigation that may require minimal modifications to the existing directory controllers relies on hardware-assisted prefetching. In this scheme, the coherence unit size is kept constant across the hierarchy. However, lower hierarchy levels prefetch larger number of coherence units on a miss than upper levels. Such scheme allows reusing of cache coherence implementations found in homogeneous machines.

The experimental results obtained in this study for inter-level miss rates among different parallelism levels confirm that there is spatial locality with respect to degree of parallelism in parallel applications, in addition to previously observed temporal locality. The differences in degrees of parallelism and memory capacity across HPAM levels motivate the use of multiple line sizes across the hierarchy as a means of reducing inter-level traffic associated with large line sizes while keeping miss rates comparable to the case where a unique line size is used across all levels An invalidation-based inter-level coherence protocol that supports such multiple line size configuration across processor-and-memory levels has been proposed, and the experimental results obtained with simulations using such protocol have confirmed that more balanced inter-level miss rate and traffic characteristics can be achieved with line sizes that increase from the top to the bottom of the hierarchy. A distribution with larger data blocks at the lowest levels of the hierarchy is consistent with the proposed HPAM organization, where lowest levels have larger amounts of memory.

Idealized caches and line sizes ranging from very small to very large have

been used in the experiments in order to observe the inherent locality behavior of the studied benchmarks. The authors believe that the overall inter-level locality behavior in systems with non-ideal caches can be derived from the results obtained.

An HPAM machine combines heterogeneity, data locality with respect to degree of parallelism and computing near memory to provide a cost-effective solution to high-performance parallel computing. The data locality studies presented in this study confirm that HPAM machines have the potential to competitively exploit the trend towards merging processor and memory technologies and the increasingly more powerful but also extremely expensive fabrication processes needed for billion-transistor chips.

Appendix A: Inter-Level Coherence Protocol Messages

```
ULW(Bj(i))  // UPPER-LVL WRITEBACK
 for all level-(j-1) sub-blocks
  if (sub-block is PART-INV) then
   ULW(sub-block)
  if (sub-block is not INV) then
   write-back sub-block from
     level j-1 to level j
   state (sub-block) = ACC
 state (Bj(i)) = ACC
```

```
LLR(Bj(i))    // LOWER-LEVELS READ
 temp = Bj(i)
 L = j
 while (temp is INV)
  increment L
  temp = level-L superblock of temp
 decrement L
 while (L >= j)
  read level-L superblock
      from level L+1
  state(level-L superblock) = ACC
  decrement L
```

```
ULI(Bj(i)) // UPPER-LVL INVALIDATE
 if (j is not the first level)
  for all level-(j-1) sub-blocks
   if (sub-block is not INV)
    state(sub-block) = INV
    ULI(sub-block)
```

```
LLP(Bj(i))  // LOWER-LEVELS
            // PARTIAL-INVALIDATE
 temp = superblock of Bj(i)
 L = j+2
 while (temp is not PART-INV)
  state(temp) = PART-INV
  increment L
  temp=level-L superblock of temp
```

```
READ(Bj(i))    // MEMORY READ
 if (Bj(i) is INV)
  LLR(Bj(i))
  state(Bj(i)) = ACC
 if (Bj(i) is PART-INV) then
  ULW(Bj(i))
 read Bj(i) from level j
```

```
WRITE(Bj(i))    // MEMORY WRITE
 if (Bj(i) is RES or ACC)
  ULI(Bj(i)); LLP(Bj(i));
 if (Bj(i) is PART-INV)
  ULW(Bj(i)); ULI(Bj(i));
 if (Bj(i) is INV)
  LLR(Bj(i)); LLP(Bj(i));
 state(Bj(i)) = RES
 write Bj(i) to level j
```

Acknowledgments: This research was supported in part by NSF grants ASC-9612133, ASC-9612023 and CDA-9617372. Renato Figueiredo is also supported by a CAPES grant.

References

1. B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, 1994.
2. D. Patterson et al. A Case for Intelligent RAM: IRAM. *IEEE Micro*, Apr 1997.
3. Figueiredo, R. J. O. and Fortes, J. A. B. and Ben-Miled, Z. and Taylor, V. and Eigenmann, R. Impact of Computing-in-Memory on the Performance of Processor-and-Memory Hierarchies. Technical Report TR-ECE-98-1, Electrical and Computer Engineering Department, Purdue University, 1998.
4. Hagersten, E. and Landin, A. and Haridi, S. DDM - A Cache-Only Memory Architecture. *IEEE Computer*, Sep. 1992.
5. J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.
6. Lenosky, D. and Laudon, J. and Gharacharloo, K. and Gupta, A. and Hennessy, J. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of the 17th Annual Int. Symp. on Computer Architecture*, May 1990.
7. M. Berry et al. The Perfect Club Benchmarks: Effective Performance Evaluation on Supercomputers. Technical Report UIUC-CSRD-827, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, July 1994.
8. M. Papamarcos and J. Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proc. of 11th Annual Int. Symp. on Computer Architecture*, 1984.
9. P. Dinda et al. The CMU Task parallel Program Suite. Technical Report CMU-CS-94-131, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, Jan 1994.
10. P.M. Kogge and T. Sunaga and H. Miyataka and K. Kitamura and E. Retter. Combined DRAM and Logic Chip for Massively Parallel Systems. *16th Conference on Advanced Research in VLSI*, 1995.
11. Standard Performance Evaluation Corporation. Spec newsletter, Sep 1995.
12. W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger and P. Tu. Parallel Programming with Polaris. *IEEE Computer*, Dec 1996.
13. Y.-S. Chen and M. Dubois. Cache Protocols with Partial Block Invalidations. In *Proc. 7th Int. Parallel Processing Symp.*, 1993.
14. Z. Ben-Miled and J.A.B. Fortes. A Heterogeneous Hierarchical Solution to Cost-efficient High Performance Computing. *Par. and Dist. Processing Symp.*, Oct 1996.
15. Z. Ben-Miled, R. Eigenmann, J.A.B. Fortes, and V. Taylor. Hierarchical Processors-and-Memory Architecture for High Performance Computing. *Frontiers of Massively Parallel Computation Symp.*, Oct 1996.